TWlib – A Generic Framework for Teamwork Applications

Reference Manual

Jürgen Lind

February 1996

Computer Science Departement
University of Kaiserslautern
AG Avenhaus

Contents

List of Figures

Chapter 1

Class Overview

This chapter is the biggest part of this report as it will describe the classes and their usage in the context of a teamwork application in detail. First of all, we will give a short outline of the classes provided by the teamwork library. Then, the various classes are discussed in detail in a bottom-up manner. The class interfaces are not included in this report in order to limit it in size. An extra reference manual is available which contains a detailed description of the class interfaces.

Figure 1.1 shows the structure of the library using two relations among the different classes: the solid arrows indicate that one class **uses the functionality** of another class (and probably the functionality of classes the second class depends on), while a dashed arrow reflects an **is-derived-from** relationship. For example, class *broadcastBuffer* uses the functionality provided by class *brStatistic* and it is derived from class *logFile*, i. e. an object of class *broadcastBuffer* has its own logging facilities.

We will now describe the class and the structure of the hierarchy tree in a few sentences to give the reader a (rough) idea of their different roles in the library, the details are discussed in the following sections.

One of the basic classes of the teamwork library is class *NumberedList* which provides a "secure" array (e. g. index range checking, automatic increment on overflow etc.). The template classes *group* and *Pairset* simplify the interface to *NumberedList* thus providing ways to handle groups of objects (similar to sets) and sets of pairs of objects of (probably) different types, respectively.

An object of class *CEPSet* fulfills administrative tasks to handle sets of connec-

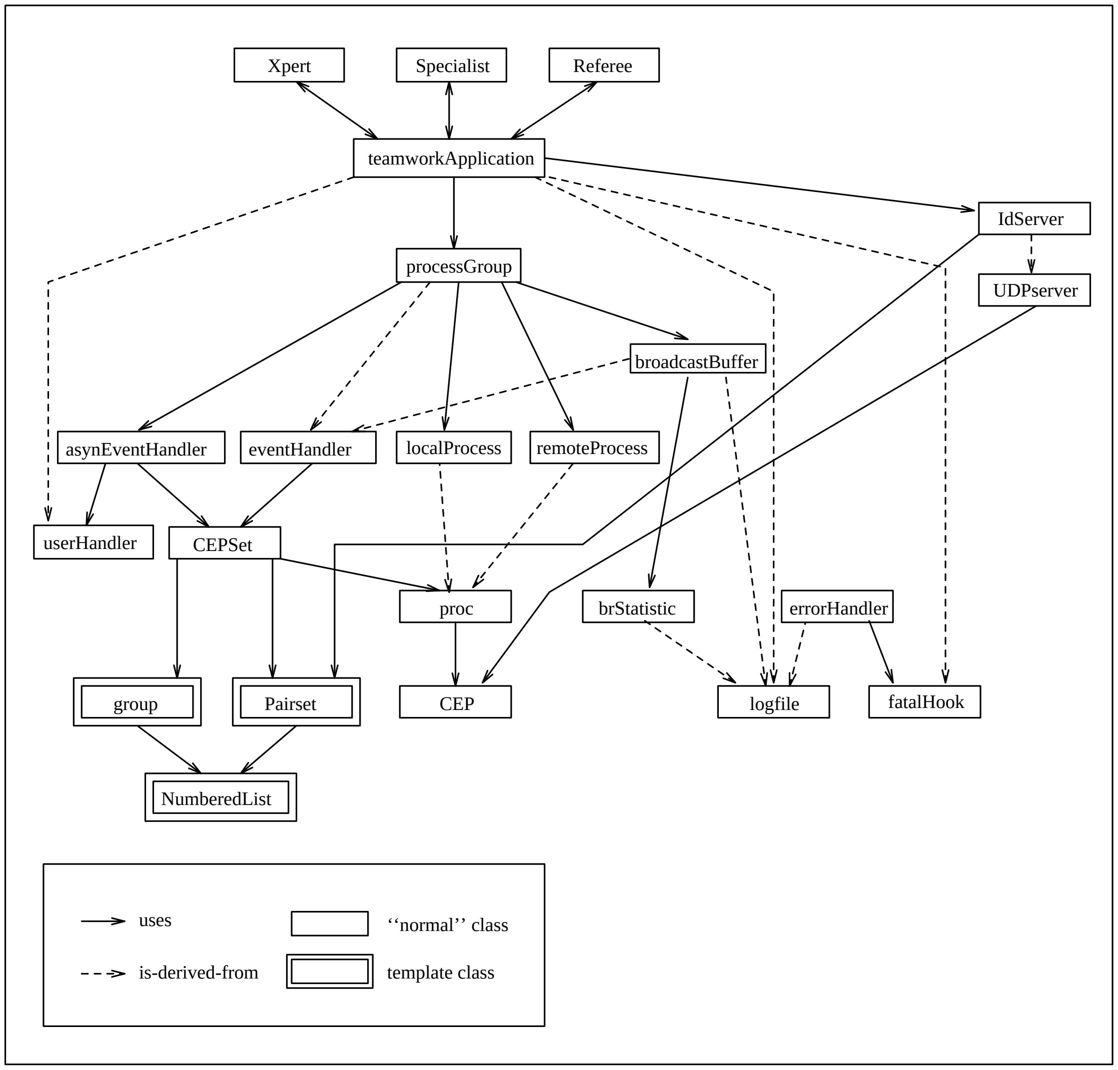

Figure 1.1: The Class Dependencies

tions a process might have to others. It uses class *proc* to identify its "owner", i. e. the process that uses the connections to communicate with others. Class *proc* itself provides methods to handle information about processes and methods to start processes on remote (or the local) host(s).

As stated in section about event handling in the general description, the event handlers use point-to-point connections to send or receive events, they have a pointer to an object of class *CEPSet* which provides an interface to these connections. As already said, this may be the same object for both handlers. Class *asynEventHandler* also uses class *userHandler* which provides a user programmed event handling routine for asynchronous events.

Class *processGroup* is an administrative class again, it holds the information a process might need about other processes. An object of class *processGroup* implements an "environment view" for a certain process. Before a process can, e. g., communicate with another (using the abstractions given by class *processGroup*), it is necessary for the other process to "join" the group. The word "join" may be a bit misleading since it does not require any physical activities by a process that wants to join a group – we will return to this problem when class *processGroup* is discussed in detail. *processGroup* depends on classes *localProcess* and *remoteProcess*, both dependent on class *proc* (see above). One of the major features of class *processGroup* is that it provides a comfortable, unified interface to all communication mechanisms supplied by the teamwork library.

Class *broadcastBuffer* contains all methods needed to offer a secure broadcasting in an ethernet environment. Class *brStatistic* stores informations gained during a broadcast session and it also provides methods to retrieve those informations, e. g. for adoption of the speed of data transmission.

errorHandler is derived from *logFile* and it contains the libraries error I/O and a hook for a user function to be called on a fatal error (e. g. to implement a clean termination of a program in the case of an unrepairable error). *errorHandler* is not indicated as being used by a specific class, since almost any of the other classes uses an object of class *errorHandler*

Class *idserver*,which is build on top of the generic base class for datagram servers, *UDPServer*, on the right hand side of figure 1.1 implements a server for unique identification numbers needed in some fields of a teamwork application.

The top level classes of our hierarchy are the generic base classes for a teamwork application (class *teamworkApplication*) and for the experts, specialists and referees used in such an application (classes *Xpert, Specialist* and *Referee*).

Class *teamworkApplication* implements the general control flow in a teamwork application, i. e. the user takes the base class and extends it by the problem specific methods. The problem specific algorithms will usually be coded in the experts, specialists and referees which are called by the *teamworkApplication* class.

This should be sufficient to give the reader a first idea of the structure of the teamwork library (and probably of the parts of a typical teamwork application). The following sections will now discuss the classes shown in figure 1.1 in detail.

Chapter 2

Class *NumberedList*

2.1 General Description

This class implements the basic data structure of our library in a generic way. “generic” means in this context, that the class is implemented as template, i. e. the user can choose which data type is used for the elements of the list. Templates are a valuable programming tool since they leave code generation (with slight differences only concerning the date type information) to the compiler.

An object of class *NumberedList* is somewhat similar to an array structure as it is used in almost any imperative programming language. The main differences between arrays and numbered lists lies within the “access security” numbered lists provide. In the C++ programming language, the programmer must take care for a lot of things when using arrays, e. g. he must check for a valid index range on element access or he must make sure that there is enough memory allocated to store the elements of an array. Other high level programming languages, such as Modula 2, already provide range checking etc., but since C++ does not, we were forced to implement these features by ourselves.

If the programmer decides to use an object of class *NumberedList* instead of an array, he is totally detached from the problems mentioned above. When a new numbered list is created, the programmer tells the compiler about the type of objects that are to be stored in the list, the initial size of the list and a so called “delta factor”. This factor comes into play when the user wants to add an element to a numbered list which is already full, i. e. all allocated memory cells are used. The system will then automatically increase the list size by the amount specified by the delta factor. To do this, a new memory region will be allocated, big enough to hold the old contents of the list as well as then additional elements. The original list is entierly copied to the new location and the old memory area (which is not needed any longer) is given back to the memory management system.

The second advantage of numbered lists is the automatic range checking. On every element access, the system looks whether the referenced index lies within the valid range (i. e. between 0 and the list size). If an invalid index is used, the system issues an error message to inform the user (programmer) about the index violation. Index range violations are very often the source of errors and we found it very helpful during our implementation of the teamwork library that those errors were detected by the system.

There are, however, some limitations and disadvantages that make numbered lists less general and usable than arrays, we will discuss some of these limitations in the reference manuals section about the implementational issues of this class. On the other hand, numbered lists are meant as extension to arrays, not a replacement. Therefore, in some situations an ordinary array may be more suitable then a *NumberedList* object.

2.2 Interface

2.2.1 Method NumberedList

Description Constructor for an object of class NumberedList¡Item¿.

Synopsis

```
#include "TWnumlist.h"

NumberedList( Index initalLength, Index aDelta )
```

Parameters

- **Index initialLength** holds the number of array elements for which memory will be reserved on creation of the object
- **Index aDelta** holds the number of elements the array will be enlarged on an overflow.

return values

none

Possible Errors

none

2.2.2 Method NumberedList

Description initializes the internal variables and allocates memory to hold `defaultNumberListSize` elements of type `Item`. `aDelta` is set to `defaultNumberListDelta`.
The default values are defined in `const.h`

Synopsis

```
#include "TWnumlist.h"

NumberedList( void )
```

Parameters

none

return values

none

Possible Errors

none

2.2.3 Method ~NumberedList

Description Deletes all elements of the array. Please note that if the items in the list are pointers, only the pointers are destroyed and **not** the objects the point to.

Synopsis

```
#include "TWnumlist.h"

 ~NumberedList( void )
```

Parameters

none

return values

none

Possible Errors

none

2.2.4 Method get

Description returns the elements with index `index` if the specified element exists or produces an error message and returns the `NULL` element for type `Item`

Synopsis

```
#include "TWnumlist.h"

Item get( Index index )
```

Parameters

Index index The index of the element the user wants to access.

return values

the specified item if it exists

`NULL` if it does not exists

Possible Errors

NumberedList::get: invalid range the index value does not point to an existing element

2.2.5 Method deleteByIndex

Description deletes the element with index `index`. If the index is not in the valid range, an error message is produced and nothing happens. Please note, that the user program can not check if the element was in the set or not.

Synopsis

```
#include "TWnumlist.h"

int deleteByIndex( Index index )
```

Parameters

Index index The index of the element that is to be deleted.

return values

`ERROR` if the specified element does not exists

`NOERROR` if the specified element was deleted

Possible Errors

NumberedList::deleteByIndex: invalid range the specified element does not exists

2.2.6 Method insert

Description Inserts the element `item` at position `index` and moves the elements between the specified position and the end of the list one position further. If the index is not in the valid range, an error message is produced and no insertion takes place. If the array is 'full', i. e. the number of elements (`count`) is equal to the size (`length`), the size is enlarged by `delta` elements.

Synopsis

```
#include "TWnumlist.h"

int insert( Index index, Item item )
```

Parameters

Index index the position of the new element

Item item the element to be inserted

return values

`ERROR` if the range was not valid

`NOERROR` if the element was inserted at the specified position

Possible Errors

NumberedList::insert: invalid range the index value was not valid

2.2.7 Method set

Description Sets the array element at position `index` to `item`. If the index is not in the valid range, an error message is produced and the function returns without any changes to the list.

Synopsis

```
#include "TWnumlist.h"

int set( Index index, Item item )
```

Parameters

Index index the position in the list that should be set

Item item the new value of the list element

return values

`ERROR` if the range was not valid

`NOERROR` if the element at the specified position was set to a new value

Possible Errors

NumberedList::set: invalid range the index value was not valid

2.2.8 Method remove

Description Removes `item` from the list. This function uses `getIndex` and `deleteByIndex`, see there for more information.

Synopsis

```
#include "TWnumlist.h"

int remove( Item item )
```

Parameters

Item item the element to be deleted

return values

see `deleteByIndex`

Possible Errors

see `deleteByIndex`

2.2.9 Method deleteAll

Description 'virtually' deletes all elements by setting `count`, i. e. the number of elements in the array, to zero. No physical deletion takes place.

Synopsis

```
#include "TWnumlist.h"

void deleteAll( void )
```

Parameters

none

return values

none

Possible Errors

none

2.2.10 Method getIndex

Description Returns the index of element `item` if the list contains that element. I not, it returns value `NotFound` of type `Index` (defined in `const.h`)

Synopsis

```
#include "TWnumlist.h"

Index getIndex( Item item )
```

Parameters

Item item the element thats index in wanted

return values

the index of `item` if `item` in the list

`NotFound` if `item` is not in the list

Possible Errors

none

2.2.11 Method append

Description Appends `item` at the end of the list by using `insert`.

Synopsis

```
#include "TWnumlist.h"

Index append( Item item )
```

Parameters

Item item the element that is to be appended

return values

the new number of elements in the list

Possible Errors

see `insert`

2.2.12 Method setLength

Description Sets the length (i. e. the maximum number of elements) of the list, the specified size may not exceed `maxNumlistSize` defined in `const.h`. If the new size would be less than the number of elements already in the list, the new size is set to the actual size. If `len` is bigger than tt maxNumlistSize len is set to that value. After these size checkings, an array of the new size is allocated and the elements of the old list are copied to the new area before the old list is deleted.

Synopsis

```
#include "TWnumlist.h"

void setLength( Index length )
```

Parameters

Index len the new size of the list

return values

none

Possible Errors

none

2.2.13 Method getCount

Description Returns the maximum index used at the moment, i. e. the number of elements in the list.

Synopsis

```
#include "TWnumlist.h"

Index getCount( void )
```

Parameters

none

return values

the size of the list

Possible Errors

none

2.3 Examples

```
class X{
```

```
  public:
    ...

  private:
    ...
};

//----------------------------------------------------------------
// templates are not automatically instantiated in our library,
// therefore we do it here...
//----------------------------------------------------------------
template class NumberedList<int>;
template class NumberedList<X*>;

void main( void )
{
  //--------------------------------------------------------------
  // create a numbered list with 10 integer values which will
  // be enlarged by 10 more elements on overflow
  //--------------------------------------------------------------
  NumberedListlist1<int> list1( 10, 10 );

  //--------------------------------------------------------------
  // the same for pointers to objects of class X
  //--------------------------------------------------------------
  NumberedListlist1<X*> list2( 10, 10 );

  for( int i = 0; i < 10; i++ ){

    list1.append( i );

    X* x;
    x = new X;
    list2.append( x );
  }

  //--------------------------------------------------------------
  // the next call will cause list1 to be moved to a new
  // location and it will be enlarged by 10 elements
  //--------------------------------------------------------------
  list1.append( 100 );
```

```
  //------------------------------------------------------------
  // the implicit call to the destructor of list2 will only
  // delete the pointers to the elements that where created with
  // new in the above loop. The objects themselves will still
  // remain in memory and if no other references exist, the
  // memory allocated for those objects will be lost since they
  // can't be freed again
  //------------------------------------------------------------
} // <- call to the destructors, list1 will be correctly deleted
  // while list2 will leave memory garbage.
```

2.4 Implementational Issues

As we have already mentioned in the general description, the are some limitations and disadvantages in the use of numbered lists. The main two are:

- Overhead: due to the additional features, we have to pay the price by accepting extra instructions the perform the enhanced functionality. Therefore, we recommend to use numbered lists in those cases, where the element access is not the critical (i. e. the most often performed) operation. For example, if one wants to implement a method which solves 500x500 linear equation systems using the Gauss algorithm, the use of numbered lists to represent the rows and columns of the matrix would be senseless and cause a lot of overhead. Why does it not make sense to use numbered lists in this case?

  1. The sizes do not change during program execution
  2. Element access errors due to index violation can only occur at the index bounds. These are only two cases that can be foreseen and therefore are easy to avoid.

  If the library user carefully decides where to use numbered lists and where not, the overhead argument against lists should not fall into account any longer.

- No possibility to offer a physical destruction of the objects kept in a numbered list. To illustrate this limitation, reconsider `list2` from the previous section: if the user wants to destroy all objects that are pointed to by the references kept in the list, it would be convenient to have method like `physicallyDeleteAll( ... )` which frees the memory at the specified locations. A function that would be able to do this work, would have to contain a line of code like `delete item`. This would work fine as long as `item` is a pointer type. If, on the other hand, `item` is a scalar type such as `int`, the compiler would try to generate code to delete an integer (not(!) an integer pointer). This is not allowed and therefore template instatiation would fail for scalar types. The solution to this problem is to create an own data type derived from *NumberedList* and implement the missing delete function according to the data type used. An example for this is the class *reportList* in the next chapter.

Chapter 3

Class *group*

3.1 General Description

The next generic data type we will present are groups. Groups can be regarded as a simplification to numbered list which only offers the most frequently used functions.

The structure of a group is very similar to the structure of a set, but there are far less operations that can be performed on a group. The main purpose of object of class *group* is to hold a collection of objects and provide an easy way to access each of the objects. As with numbered lists, the user can chose the type of the objects in the group, because *group* is implemented as template class.

3.2 Interface

3.2.1 Method group

Description Constructor for an object of class *group*. The objects of this class are implemented numbered lists.

Synopsis

```
#include "TWgroup.h"

group( void )
```

Parameters

none

return values

none

Possible Errors

none

3.2.2 Method ~group

Description Detructor, deletes all members.

Synopsis

```
#include "TWgroup.h"

~group( void )
```

Parameters

none

return values

none

Possible Errors

none

3.2.3 Method clear

Description Deletes all members of a group.

Synopsis

```
#include "TWgroup.h"

void clear( void )
```

Parameters

none

return values

none

Possible Errors

see `NumberedList::deleteAll`

3.2.4 Method add

Description Adds a new member to the group

Synopsis

```
#include "TWgroup.h"

void add( groupType mem )
```

Parameters

groupType mem the new group member

return values

none

Possible Errors

see `NumberedList::append`

3.2.5 Method remove

Description Removes a member from the group

Synopsis

```
#include "TWgroup.h"

void remove( groupType mem )
```

Parameters

groupType mem

return values

none

Possible Errors

see `NumberedList::append`

3.2.6 Method set

Description Sets the group member with the given index to the specified value.

Synopsis

```
#include "TWgroup.h"

void set( Index index, groupType mem )
```

Parameters

Index index

groupType mem

return values

none

Possible Errors

see `NumberedList::set`

3.2.7 Method get

Description Returns the group member with the given index.

Synopsis

```
#include "TWgroup.h"

groupType get( Index index )
```

Parameters

Index index

return values

the specified member if it exists

`NULL` if it does not exists

Possible Errors

see `NumberedList::get`

3.2.8 Method groupSize

Description Returns the actual group size.

Synopsis

```
#include "TWgroup.h"

Index groupSize( void )
```

Parameters

none

return values

the group size

Possible Errors

none

3.2.9 Method isMember

Description Checks whether the given object is in the group or not

Synopsis

```
#include "TWgroup.h"

bool isMember( groupType mem )
```

Parameters

groupType mem the object to test

return values

true `mem` is a group member

flase `mem` is not a group member

Possible Errors

see `NumberedList::getIndex`

3.2.10 Method operator =

Description Assigns the elements of another group to this group.
NOTE: This implies the deletion of all elements of the destination group.

Parameters

const group¡groupType¿& gr the source group

return values

a pointer to this object

Possible Errors

see `NumberedList::getCount`, `NumberedList::get`

3.2.11 Method print

Description Prints the contents of the group in the following form:

```
(item_1, item_2, ...item_n)
```

Synopsis

```
#include "TWgroup.h"

void  print( void )
```

Parameters

none

return values

none

Possible Errors

see `NumberedList::getCount`, `NumberedList::get`

3.3 Examples

```
//----------------------------------------------------------------
// templates are not automatically instantiated in our library,
// therefore we do it here...
//----------------------------------------------------------------
template class group<int>;

//----------------------------------------------------------------
// create a new data type...
//----------------------------------------------------------------
typedef group<int> intGroup

void main( void )
{
  //--------------------------------------------------------------
  // create an object of the new type
  //--------------------------------------------------------------
  intGroup IG;

  IG.add( 4 );
  IG.add( 2 );
  IG.add( 5 );
  IG.add( 4 );

  IG.remove( 2 );

  for( int i = 0; i < IG.groupSize(); i++ )
    cout << IG.get( i ) << " ";
  //-------------------------------------------------------------
  // output will be '4 5 4'
```

```
  //------------------------------------------------------------

  IG.remove( 4 );

  IG.print();
  //-----------------------------------------------------------
  // output will be '5 4'
  //------------------------------------------------------------
}
```

3.4 Implementational Issues

Chapter 4

Class *Pairset*

4.1 General Description

Another basic data structure which is often needed in any kind of application are pairsets. As the name already suggests, the elements stored in the set structure are two-element-objects called "pairs". Please note, that a set in the mathematical sense does not imply any ordering among its elements. Therefore, it is not possible to go through a pairset element-by-element. To access to a certain object in the pairset, the user must know at least one element of a pair.

Pairsets are often called associative arrays to express the fact, that the grouping into pairs represents a certain relationship between the two elements. We will, however, stick to our term and speak of pairsets, mainly because the term "associative array" may suggest a normal array access to the set elements – which does not hold for our implementation.

The operations which can be performed on pairsets are very simple and self explaining, the examples presented later should enable the reader to use pairsets without any difficulties in his own applications. As with numbered lists, class *Pairset* is implemented as template and therefore its very flexible and usable for all kinds of applications.

4.2 Interface

4.2.1 Method Pairset

Description Creates a new object of class *Pairset* using two objects of class *NumberedList*. The values for `initialPairsetSize` and `increasePairset` are defined in `const.h`.

Synopsis

```
#include "TWpairset.h"

Pairset( void )
```

Parameters

none

return values

none

Possible Errors

none

4.2.2 Method ~Pairset

Description Destructor for a pairset, deletes the two numbered lists which where used to implement the pairset

Synopsis

```
#include "TWpairset.h"

~Pairset( void )
```

Parameters

none

return values

none

Possible Errors

none

4.2.3 Method clear

Description Removes all elements from the pairset

Synopsis

```
#include "TWpairset.h"

void clear( void )
```

Parameters

none

return values

none

Possible Errors

see `NumberedList::deleteAll`

4.2.4 Method enter

Description Places a new pair `<item1,item2>` into the pairset. The two items must get the same number in their respective lists, this is important for all further actions and must be taken into account if class *NumberedList* is reimplemented.

Synopsis

```
#include "TWpairset.h"

void enter( type1 item1, type2 item2 )
```

Parameters

type1 item1 the first element of the new pair

type2 item2 the second element of the new pair

return values

none

Possible Errors

see `NumberedList::append`

4.2.5 Method removeByFirst

Description Removes a pair from the pairset using the first element of each pair to see if the pair must be removed. Please note, that if multiple pairs with the same first element exist, only one of them is removed.

Synopsis

```
#include "TWpairset.h"

void removeByFirst( type1 item )
```

Parameters

type1 item the first element of the pair that should be removed

return values

none

Possible Errors

see `NumberedList::getIndex`, `NumberedList::remove`, `NumberedList::deleteByIndex`

4.2.6 Method removeBySecond

Description Removes a pair from the pairset using the second element of each pair to see if the pair must be removed. Please note, that if multiple pairs with the same second element exist, only one of them is removed.

Synopsis

```
#include "TWpairset.h"

void removeBySecond( type2 item )
```

Parameters

type2 item the second element of the pair that should be removed

return values

none

Possible Errors

see `NumberedList::getIndex`, `NumberedList::remove`, `NumberedList::deleteByIndex`

4.2.7 Method getByFirst

Description Returns the second element of a pair with first element `item`.

Synopsis

```
#include "TWpairset.h"

type2 getByFirst( type1 item )
```

Parameters

type1 item the first element of the desired pair

return values

the second element of the pair if an element with matching first part is found

`NULL` **value of type2** if no matching pair is found

Possible Errors

see `NumberedList::getIndex`, `NumberedList::get`

4.2.8 Method getBySecond

Description Returns the first element of a pair with second element `item`.

Synopsis

```
#include "TWpairset.h"

type1 getBySecond( type2 item )
```

Parameters

type2 item the second element of the desired pair

return values

the first element of the pair if an element with matching second part is found

`NULL` **value of type1** if no matching pair is found

Possible Errors

see `NumberedList::getIndex`, `NumberedList::get`

4.2.9 Method getFirsts

Description Returns a goup with the first elements of each pair

Synopsis

```
#include "TWpairset.h"

void getFirsts( group<type1>* gr )
```

Parameters

group¡type1¿* gr the goup which should be filled with the first elements

return values

none

Possible Errors

see `NumberdList::getCount`, `group::add`

4.2.10 Method getSeconds

Description Returns a goup with the second elements of each pair

Synopsis

```
#include "TWpairset.h"

void getSeconds( group<type2>* gr )
```

Parameters

group¡type2¿* gr the goup which should be filled with the second elements

return values

none

Possible Errors

see `NumberdList::getCount`, `group::add`

4.2.11 Method getSize

Description Returns the size of the pairset.

Synopsis

```
#include "TWpairset.h"

int getSize( void )
```

Parameters

none

return values

the size of the pairset

Possible Errors

[see `NumberdList::getCount`

4.2.12 Method pack

Description Packs the contents of the pairset into a char buffer `buf` with size `size`. If the contents of a set are bigger than the size of the buffer, `pack` returns with an error and without completing the packing.

Synopsis

```
#include "TWpairset.h"

int pack( char* buf, int size )
```

Parameters

char* buf the destination of the packed set

int size the size of the destination area

return values

`ERROR` the destination area is too small

`NOERROR` the set was packed to the specified location

Possible Errors

Pairset::pack: not enought memory

see **also:** `NumberedList::getCount`, `NumberedList::get`

4.2.13 Method unpack

Description Unpacks the contents of the pairset from a char buffer `buf` with size `size`.

Synopsis

```
#include "TWpairset.h"

void unpack( char* buf, int size )
```

Parameters

char* buf the address of the buffer containing packed set

int size the size of the buffer

return values

none

Possible Errors

see `NumberedList::deleteAll`, `NumberedList::append`

4.2.14 Method print

Description prints the contents of the pairset in the following form:

```
(item1, item2)
(item1, item2)
      :
      :
(item1, item2)
```

Synopsis

```
#include "TWpairset.h"

void print( void )
```

Parameters

none

return values

none

Possible Errors

see `NumberedList::getCount`, `NumberedList::get`

4.3 Examples

```
//---------------------------------------------------------------
// templates are not automatically instantiated in our library,
// therefore we do it here...
//---------------------------------------------------------------
template class group<int>;
template class Pairset<int, int>;

void main( void )
{
  //-------------------------------------------------------------
  // create a new pairset for pairs of ints
  //-------------------------------------------------------------
  Pairset<int, int> ps;

  ps.enter( 1, 5 );
  ps.enter( 4, 6 );
  ps.enter( 6, 1 );
  ps.enter( 7, 9 );
```

```
  ps.print(); // output see below

  ps.deleteByFirst( 4 );

  ps.print(); // output see below

  ps.deletebySecond( 9 );

  ps.print(); // output see below

  group<int> gr;

  ps.getFirsts( &gr );
  for( int i = 0; i < gr.groupSize(); i++ ){
    cout << gr.get( i ) << endl; // output see below
  }
}
```

Now, let's see what this sample program will produce as output:

1. will look like

   ```
   ( 1, 5 )
   ( 4, 6 )
   ( 6, 1 )
   ( 7, 9 )
   ```

2. will be

   ```
   ( 1, 5 )
   ( 6, 1 )
   ( 7, 9 )
   ```

3. is

   ```
   ( 1, 5 )
   ( 6, 1 )
   ```

4. and last

1
6

4.4 Implementational Issues

Chapter 5

Class *errorHandler*

5.1 General Description

In any application – distributed or not – errors can occur. These errors can be due to wrong design or implementation, invalid user input or failures in operating system calls. To avoid as much damage as possible, any error should be caught and reported to the programmer to enable him, to handle an error situation in the best possible way. When designing a library, the error handling mechanism becomes even more difficult, because the library supplier can not know what is the appropriate action in an error situation. The appropriate action is almost always application dependent and can therefore not be foreseen by the library programmer. Because of this, the teamwork library has a global object `ERR` of class *errorHandler* to which any errors occurring during a teamwork application will be reported. The *errorHandler* object will then inform the library user that something went wrong. To inform the user program that an error occurred, most of the teamwork library functions will return an error code to indicate that the action could not be completed properly. An error recovery is (and must be) left to the programmer except for the cases where the error is so serious that the application can not continue and must therefore be aborted.

In the teamwork library, we have classified the possible errors with respect to three attributes:

Possible error types

- Internal errors of the teamwork library. This type also includes error which occur because of wrong usage of the library by a user program.
- Errors which occur as a consequence of an operating system call.

Possible error consequences

- Inform the user and the user program that something went wrong (error indication).

- Terminate the application as a consequence of an unrecoverable error.

Possible error notification

- Issue an error message. The message is only meant to inform the (application) user, the user program itself will only get an error notification without further details of what happened.
- Set an internal error status which will contain additional information about what happened. The internal status can be queried by a user program.

Please note, that class *errorHandler* is derived from class *logFile*, i. e. any error messages can be redirected to a file etc. The default output descriptor for error messages is `stdout`. For an example of error message redirection, refer to section 9.3.

According to this classification, we have implemented eight error report methods, which are again subdivided in four groups. Each group contains two methods which take either an (ASCII) error message to be printed or an error code as parameter. In the first case, the message is printed on the error output descriptor. These methods can be used to inform the user. The second way to report an error will not only print an error message, but it will also set the internal error status. This method should be used when additional information about the reason for the error is available. The message which is printed on the error channel corresponds to the error code. The error codes are defined in file `error.h`, the messages in file `error.cc` in the `src` directory of the library distribution.

The groups of error report methods are:

Method `log` will only print an error message (either specified by the message string or by the error code).

Method `sysLog` will print the specified error message and also the error message of the operating system indicated by `errno`

Method `fatalLog` will print an error message and then terminate the application.

Method `fatalSysLog` will print the specified message, the operating systems message, and then terminate the application.

Besides the methods for error reporting, class *errorHandler* has an additional feature. This is the possibility to install a so called **fatal hook**. The object which is announced to the library as fatal hook must be derived from class *fatalHook* and it should override the member function `abnormalTermination` of the base class. When one of the `fatal...` methods of the above list is called, the library will call the `abnormalTermination` method of the *fatalHook* object before terminating the application. For example, a fatal hook can be used to terminate all processes of a distributed application if an unrecoverable error occurs in one of the processes in order to guarantee a “clean” termination of the entire application. Refer to section 14 and the library code for class *teamworkApplication* to see how a fatal hook can be used.

Please note, that class *errorHandler* also installs a signal handler for the `SIGINT` signal. If the application program user hits *Ctrl-C* on his keyboard, the library will call the fatal hook (if installed) and then terminate the process to which the break signal was sent.

5.2 Interface

5.2.1 Method errorHandler

Description Constructor for an object of class *errorHandler*.

Synopsis

```
#include "TWerrorHandler.h"

errorHandler( string fname )
```

Parameters

string fname the name of the file into which the error messages will be written

return values

none

Possible Errors

none

5.2.2 Method errorHandler

Description Constructor for an object of class *errorHandler*, the error messages will be send to `stderr`.

Synopsis

```
#include "TWerrorHandler.h"

errorHandler( void )
```

Parameters

none

return values

none

Possible Errors

none

5.2.3 Method ~errorHandler

Description Destructor for an object of class *errorHandler*.

Synopsis

```
#include "TWerrorHandler.h"

~errorHandler( void )
```

Parameters

none

return values

none

Possible Errors

none

5.2.4 Method open

Description Redirects the error message which would be sent to `stderr` into the file with the specified name. A call to this method is only valid when the constructor without parameters was used for the object creation.

Synopsis

```
#include "TWerrorHandler.h"

void open( string filename )
```

Parameters

string fname the name of the file into which the error message should be written

return values

none

Possible Errors

none

5.2.5 Method redirect

Description Redirects the error message which would be sent to specified descriptor into the file with the given name.

Synopsis

```
#include "TWerrorHandler.h"

void redirect( string fname, FILE* redir )
```

Parameters

string fname the name of the file into which the error message should be written

FILE* redir the descriptor of the output stream which should be redirected

return values

none

Possible Errors

none

5.2.6 Method close

Description Writes an end message on the descriptor used for error logging and closes it.

Synopsis

```
#include "TWerrorHandler.h"

void close( void )
```

Parameters

none

return values

none

Possible Errors

invalid descriptor for logging, exiting ... the descriptor associated with the *errorHandler* object is not valid. This can be due to an error during an `open` or `redirect` call.

5.2.7 Method log

Description Writes an error message to the associated descriptor.

Synopsis

```
#include "TWerrorHandler.h"

void log( string where, string format ... )
```

Parameters

string where the name of the file where the error occurred

string format the format string (the same as for `printf`) for the error message

... var args a variable number of following arguments of any type which will form the error message

return values

none

Possible Errors

invalid descriptor for logging, exiting ... the descriptor associated with the *errorHandler* object is not valid. This can be due to an error during an `open` or `redirect` call.

5.2.8 Method log

Description Writes an error message to the associated descriptor.

Synopsis

```
#include "TWerrorHandler.h"

void log( string where, errCode ec )
```

Parameters

string where the name of the file where the error occurred

errCode ec the error code of the error. The various error code are defined in `error.h`, the associated error message are defined in `errMsg[]` in `error.cc`.

return values

none

Possible Errors

invalid descriptor for logging, exiting ... the descriptor associated with the *errorHandler* object is not valid. This can be due to an error during an `open` or `redirect` call.

5.2.9 Method sysLog

Description Writes an error message to the associated descriptor. The operating systems error message is also written to that descriptor.

Synopsis

```
#include "TWerrorHandler.h"

void sysLog( string where, string format ... )
```

Parameters

string where the name of the file where the error occurred

string format the format string (the same as for `printf`) for the error message

... var args a variable number of following arguments of any type which will form the error message

return values

none

Possible Errors

invalid descriptor for logging, exiting ... the descriptor associated with the *errorHandler* object is not valid. This can be due to an error during an `open` or `redirect` call.

5.2.10 Method sysLog

Description Writes an error message to the associated descriptor. The operating systems error message is also written to that descriptor.

Synopsis

```
#include "TWerrorHandler.h"

void sysLog( string where, errCode ec )
```

Parameters

string where the name of the file where the error occurred

errCode ec the error code of the error. The various error code are defined in `error.h`, the associated error message are defined in `errMsg[]` in `error.cc`.

return values

none

Possible Errors

invalid descriptor for logging, exiting ... the descriptor associated with the *errorHandler* object is not valid. This can be due to an error during an `open` or `redirect` call.

5.2.11 Method fatalLog

Description Writes an error message to the associated descriptor. After the message is written to the descriptor, the system calls the user handler for abnormal terminations (if installed) and the the program exists.

Synopsis

```
#include "TWerrorHandler.h"

void fatalLog( string where, string format ... )
```

Parameters

string where the name of the file where the error occurred

string format the format string (the same as for `printf`) for the error message

... var args a variable number of following arguments of any type which will form the error message

return values

none

Possible Errors

invalid descriptor for logging, exiting ... the descriptor associated with the *errorHandler* object is not valid. This can be due to an error during an `open` or `redirect` call.

5.2.12 Method fatalLog

Description Writes an error message to the associated descriptor. After the message is written to the descriptor, the system calls the user handler for abnormal terminations (if installed) and the the program exists.

Synopsis

```
#include "TWerrorHandler.h"

void fatalLog( string where, errCode ec )
```

Parameters

string where the name of the file where the error occurred

errCode ec the error code of the error. The various error code are defined in `error.h`, the associated error message are defined in `errMsg[]` in `error.cc`.

return values

none

Possible Errors

invalid descriptor for logging, exiting ... the descriptor associated with the *errorHandler* object is not valid. This can be due to an error during an `open` or `redirect` call.

5.2.13 Method fatalSysLog

Description Writes an error message to the associated descriptor. The operating systems error message is also written to that descriptor. After the message is written to the descriptor, the system calls the user handler for abnormal terminations (if installed) and the the program exists.

Synopsis

```
#include "TWerrorHandler.h"

void fatalSysLog( string where, string format ... )
```

Parameters

string where the name of the file where the error occurred

string format the format string (the same as for `printf`) for the error message

... var args a variable number of following arguments of any type which will form the error message

return values

none

Possible Errors

invalid descriptor for logging, exiting ... the descriptor associated with the *errorHandler* object is not valid. This can be due to an error during an `open` or `redirect` call.

5.2.14 Method fatalSysLog

Description Writes an error message to the associated descriptor. The operating systems error message is also written to that descriptor. After the message is written to the descriptor, the system calls the user handler for abnormal terminations (if installed) and the the program exists.

Synopsis

```
#include "TWerrorHandler.h"

void fatalSysLog( string where, errCode ec )
```

Parameters

string where the name of the file where the error occurred

errCode ec the error code of the error. The various error code are defined in `error.h`, the associated error message are defined in `errMsg[]` in `error.cc`.

return values

none

Possible Errors

invalid descriptor for logging, exiting ... the descriptor associated with the *errorHandler* object is not valid. This can be due to an error during an `open` or `redirect` call.

5.2.15 Method getStatus

Description Returns the current error status.

Synopsis

```
#include "TWerrorHandler.h"

errCode getStatus( void )
```

Parameters

none

return values

the current error status

Possible Errors

none

5.2.16 Method setFatalHook

Description Sets the user handler to call in the case of an abnormal program termination. The object whichs address is passed to this method must be derived from class *fatalHook*.

Synopsis

```
#include "TWerrorHandler.h"

void setFatalHook( fatalHook* fh )
```

Parameters

fatalHook* fh a pointer to the user object to call in the case of an abnormal program termination

return values

none

Possible Errors

none

5.2.17 Method installSigHndl

Description Installs the signal handlers for some signals which would cause a core dump and a program termination when not caught. These signals are `SIGILL`, `SIGBUS` and `SIGSEGV`. Refer to the manual pages for `signal` for detailed informations on signals.

Synopsis

```
#include "TWerrorHandler.h"

void installSigHndl( void )
```

Parameters

none

return values

none

Possible Errors

none

5.2.18 Method breakHandler

Description This method is called when the user types 'Ctrl-C' on his keyboard. The system will then call the user handler for abnormal terminations (if installed) and the the program exists.

Synopsis

```
#include "TWerrorHandler.h"

static void breakHandler( void )
```

Parameters

none

return values

none

Possible Errors

none

5.2.19 Method catchSignal

Description This method is called when a signal which would cause a core dump and a program termination when not caught occurs. The system will then call the user handler for abnormal terminations (if installed) and the the program exists.

Synopsis

```
#include "TWerrorHandler.h"

static void catchSignal( void )
```

Parameters

none

return values

none

Possible Errors

none

5.2.20 Method getTime

Description Writes the current time in humanly readable form into the specified string.

Synopsis

```
#include "TWerrorHandler.h"

void getTime( string output )
```

Parameters

string output the destination string for the time value.

return values

none

Possible Errors

none

5.2.21 Method printInit

Description Prints the initial message for an error logfile.

Synopsis

```
#include "TWerrorHandler.h"

void printInit( void )
```

Parameters

none

return values

none

Possible Errors

none

5.3 Examples

5.4 Implementational Issues

Chapter 6

Class *CEP*

6.1 General Description

An object of class *CEP* (which stands for communication endpoint) is merely a (sometimes enhanced) interface to the use of socket. The reader familiar with socket communication based on Berkley sockets (see [Ste90] or [CS93]) should have no problems in understanding the methods of this class.

Each object of this class has an associated socket descriptor it operates on. This can be the descriptor of a datagram as well as of a stream socket. The methods that can be performed on the socket depend on the respective type. When a CEP-object is created, the user can choose if he wants to use an already existing socket or if he wants to create a new one. If he wants to create an new socket, the type of the socket is not chosen when creating the object. A subsequent call to either `datagram` or `PtP` (point-to-point) has to be executed to set the object to a concrete type.

6.2 Interface

6.2.1 Method CEP

Description Constructor for an object of class *CEP*. This function assigns an already existing socket descriptor to the newly created object. All of the following actions will be performed on that socket.

Synopsis

```
#include "TWCEP.h"

CEP( t_socket s )
```

Parameters

t_socket s the socket which will become the new communication endpoint

return values

none

Possible Errors

none

6.2.2 Method CEP

Description Constructor for an object of class *CEP*. Since no existing socket is is assigned, this constructor implies a subsequent call to `PtP`, `datagram` or `connect` must be done before any further action can be taken.

Synopsis

```
#include "TWCEP.h"

CEP( void )
```

Parameters

none

return values

none

Possible Errors

none

6.2.3 Method ~CEP

Description Destructor for an object of class *CEP*. Dummy function

Synopsis

```
#include "TWCEP.h"

~CEP( void )
```

Parameters

none

return values

none

Possible Errors

none

6.2.4 Method datagram

Description A new datagram socket (`SOCK_DGRAM`) is created and bound to the given port number.

Synopsis

```
#include "TWCEP.h"

int datagram( int port )
```

Parameters

int port the port number which will be assigned to the new datagram socket

return values

`ERROR` if the socket could not be created

`NOERROR` if the socket was created and assigned the specified port number

Possible Errors

CEP::datagram: unable to create socket the system call to `sock` failed, `errno` should be checked for further informations

CEP::datagram: error in bind the system call to `bind` failed, `errno` should be checked for further informations

6.2.5 Method datagram

Description A new datagram socket (`SOCK_DGRAM`) is created. The port number for the new socket is assigned by the operation system.

Synopsis

```
#include "TWCEP.h"

int datagram( void )
```

Parameters

none

return values

`ERROR` if the socket could not be created

the port number of the new socket if the socket was created successfully

Possible Errors

CEP::datagram: unable to create socket the system call to `sock` failed, `errno` should be checked for further informations

CEP::datagram: error in bind the system call to `bind` failed, `errno` should be checked for further informations

CEP::datagram: error in getsockname the system call to `getsockname` failed, `errno` should be checked for further informations

6.2.6 Method PtP

Description A new stream socket (`SOCK_STREAM`) is created and bound to the given port number.

Synopsis

```
#include "TWCEP.h"

int PtP( int port )
```

Parameters

int port the port number which will be assigned to the new stream socket

return values

`ERROR` if the socket could not be created

`NOERROR` if the socket was created and assigned the specified port number

Possible Errors

CEP::PtP: unable to create socket the system call to `sock` failed, `errno` should be checked for further informations

CEP::PtP: error in bind the system call to `bind` failed, `errno` should be checked for further informations

6.2.7 Method PtP

Description A new stream socket (`SOCK_STREAM`) is created. The port number for the new socket is assigned by the operation system.

Synopsis

```
#include "TWCEP.h"

int PtP( void )
```

Parameters

none

return values

`ERROR` if the socket could not be created

The port number of the new socket if the socket was created

Possible Errors

CEP::PtP: unable to create socket the system call to `sock` failed, `errno` should be checked for further informations

CEP::PtP: error in bind the system call to `bind` failed, `errno` should be checked for further informations

CEP::PtP: error in getsockname the system call to `getsockname` failed, `errno` should be checked for further informations

6.2.8 Method accept

Description Accepts a connection. Please note, that a call to the `accept` function of the operating system may be interrupted by a signal and that it is not resumed after the signal is processed. This function performs repeated calls to `accept` if the call is interrupted by signal.
IMPORTANT: This method may block the system if no connection can be established.

Synopsis

```
#include "TWCEP.h"

t_socket accept( void )
```

Parameters

none

return values

`INVALID` if an error occurred, see `const.h` for the definition of `INVALID`

a new socket descriptor if the a connection was established

Possible Errors

CEP::accept: error an error occured in the operating systems `accept`, `errno` should be checked for further informations

6.2.9 Method accept

Description Accepts a connection. Please note, that a call to the `accept` function of the operating system may be interrupted by a signal and that it is not resumed after the signal is processed. This function performs repeated calls to `accept` if the call is interrupted by signal.

If no connection is established within the given time, `accept` returns with an error code. In this case, the global variable `err` is set to `TIMEOUT`.

Synopsis

```
#include "TWCEP.h"

t_socket accept( int timeout )
```

Parameters

int timeout the time to wait for an incoming connection, measured in seconds

return values

`INVALID` if an error occurred, `err` should be checked to see if the `accept` call timed out

a new socket descriptor if a connections was established

Possible Errors

CEP::accept: select timed out the `select` call timed out

CEP::accept: error in select an error occurred in the operating systems `select` function, `errno` should be checked for further informations

CEP::accept: error an error occurred in the operating systems `accept`, `errno` should be checked for further informations

6.2.10 Method connect

Description This method tries to establish a connection to another process. The port number and the hostname where that process resides are stored in the `procInfo` structure (see `types.h` for further description). Within `connect`, a new socket is created and then connected to the peer process if possible.

Synopsis

```
#include "TWCEP.h"

int connect( procInfo *pi )
```

Parameters

procInfo *pi a pointer to a structure which contains all necessary informations to connect to another process (see `types.h`)

return values

`ERROR` if an error occurred

`NOERROR` if a connections was established.

Possible Errors

CEP::connect: unable to create socket an error occurred in the operating systems `socket` function, `errno` should be checked for further informations

CEP::connect: no such host the operating system call to `gethostbyname` returned an empty `hostent` structure

CEP::connect: error an error occurred in the operating systems `connect` function, `errno` should be checked for further informations

6.2.11 Method getSock

Description Returns the socket descriptor which is associated with the CEP.

Synopsis

```
#include "TWCEP.h"

t_socket getSock( void )
```

Parameters

none

return values

INVALID if the CEP is not associated with a socket

the socket descriptor if the CEP is associated with a socket

Possible Errors

none

6.2.12 Method read

Description A specified amount of bytes are read from the associated socket and copied to a destination address in memory. There may be multiple calls to the operating systems `read` function because that function may return without reading the entier amount of data (see manual pages for `read`(2)).

Synopsis

```
#include "TWCEP.h"

bool read( void* data, int len )
```

Parameters

void *data the destination of the read data

int len the size of the destination buffer

return values

false if an error occurred

true if the data was successfully read

Possible Errors

CEP::read: error an error occurred in the operating systems `read` function, `errno` should be checked for further informations

6.2.13 Method readOOB

Description A specified amount of out-of band data is read from the associated socket and copied to a destination address in memory. There may be multiple calls to the operating systems `recv` function because that function may return without reading the entier amount of data (see manual pages for `recv`).
Please note, that the socket must be marked as socket for urgent data before out-of-band messages can be received, see the manual pages for `fcntl` for further information.

Synopsis

```
#include "TWCEP.h"

bool readOOB( void* data, int len )
```

Parameters

void *data the destination of the read data

int len the size of the destination buffer

return values

false if an error occurred

true if the data was successfully read

Possible Errors

CEP::readOOB: error an error occurred in the operating systems `recv` function, `errno` should be checked for further informations

6.2.14 Method write

Description A specified amount of bytes are written to the associated socket. There may be multiple calls to the operating systems `write` function because that function may return without writing the entier amount of data (see manual pages for `write`(2)).

Synopsis

```
#include "TWCEP.h"

bool write( void* data, int len )
```

Parameters

void *data the address of the data to be written

int len the size of the source buffer

return values

false if an error occurred

true if the data was successfully written

Possible Errors

CEP::write: error an error occurred in the operating systems `write` function, `errno` should be checked for further informations

6.2.15 Method writeOOB

Description A specified amount of out-of-band data is written to the associated socket. There may be multiple calls to the operating systems `send` function because that function may return without writing the entier amount of data (see manual pages for `send`).
Please note, that the socket must be marked as socket for urgent data before out-of-band messages can be received, see the manual pages for `fcntl` for further information.

Synopsis

```
#include "TWCEP.h"

bool writeOOB( void* data, int len )
```

Parameters

void *data the address of the data to be written

int len the size of the source buffer

return values

false if an error occurred

true if the data was successfully written

Possible Errors

CEP::write: error an error occurred in the operating systems `send` function, `errno` should be checked for further informations

6.2.16 Method recvDgram

Description A datagram is received on the associated socket and the address of the sender is returned if desired. If there is no pending data, the method will return instantly. This method is a front-end to the systems `recvfrom` function, refer to the manual pages for further information.

Synopsis

```
#include "TWCEP.h"

int recvDgram( void* data, int len, struct sockaddr* from, int* addrLen )
```

Parameters

void* data the destination address of the datagram

int len the size of the destination buffer

struct sockaddr* from if not `NULL`, the address of the sender will be placed at the specified address

int* addrLen value-result parameter, if `from` is not `NULL`, it must contain the size of the buffer associated with `from`. On return, it will contain the size of sender address.

return values

`ERROR` if an error occurred

number of bytes received if a datagram was read

Possible Errors

CEP::recvDgram: error an error occurred in the operating systems `recvfrom` function, `errno` should be checked for further informations

6.2.17 Method waitDgram

Description A datagram is received on the associated socket and the address of the sender is returned if desired. If no data can be read within a specified amount of time, the method returns with an error message. In this case, `err` will be set to indicate the timeout.

Synopsis

```
#include "TWCEP.h"

int waitDgram( void* data, int len, int timeout, struct sockaddr* from, int* addrLen )
```

Parameters

void* data the destination address of the datagram

int len the size of the destination buffer

int timeout the time to wait for a datagram, measured in seconds

struct sockaddr* from if not `NULL`, the address of the sender will be placed at the specified address

int* addrLen value-result parameter, if `from` is not `NULL`, it must contain the size of the buffer associated with `from`. On return, it will contain the size of sender address.

return values

`ERROR` if an error occurred

number of bytes received if a datagram was read

Possible Errors

CEP::waitDgram: select timed out the `select` call timed out

CEP::waitDgram: error in select an error occurred in the operating systems `select` function, `errno` should be checked for further informations

6.2.18 Method waitDgram

Description A datagram is received on the associated socket and the address of the sender is returned if desired. If there is no pending data, this method will block until some data arrives.

Synopsis

```
#include "TWCEP.h"

int waitDgram( void* data, int len, struct sockaddr* from, int* addrLen )
```

Parameters

void* data the destination address of the datagram

int len the size of the destination buffer

struct sockaddr* from if not `NULL`, the address of the sender will be placed at the specified address

int* addrLen value-result parameter, if `from` is not `NULL`, it must contain the size of the buffer associated with `from`. On return, it will contain the size of sender address.

return values

`ERROR` if an error occurred

number of bytes received if a datagram was read

Possible Errors

CEP::waitDgram: error in select an error occurred in the operating systems `select` function, `errno` should be checked for further informations

6.2.19 Method sendDgram

Description Sends a datagram to the specified port on the given host.

Synopsis

```
#include "TWCEP.h"

int sendDgram( void* data, int len, char* to, int port )
```

Parameters

void* data address of the data which should be sent as datagram

int len size of the data

char* to pointer to the hostname of the destination host

int port port number on the destination host

return values

`ERROR` if an error occurred

number of bytes written if a datagram was sent

Possible Errors

CEP::sendDgram: no such host the operating system call to `gethostbyname` returned an empty `hostent` structure

CEP::sendDgram: error in sendto an error occurred in the operating systems `sendto` function, `errno` should be checked for further informations

6.2.20 Method sendDgram

Description Sends a datagram to an address specified in TCP/IP format.

Synopsis

```
#include "TWCEP.h"

int sendDgram( void* data, int len, struct sockaddr* to, int toLen )
```

Parameters

void* data address of the data which should be sent as datagram

int len size of the data

struct sockaddr* to the destination address

int toLen the length of the destination address

return values

`ERROR` if an error occurred

number of bytes written if a datagram was sent

Possible Errors

CEP::sendDgram: no such host the operating system call to `gethostbyname` returned an empty `hostent` structure

CEP::sendDgram: error in sendto an error occurred in the operating systems `sendto` function, `errno` should be checked for further informations

6.2.21 Method close

Description Closes the associated socket.

Synopsis

```
#include "TWCEP.h"

void close( void )
```

Parameters

none

return values

none

Possible Errors

CEP::close: error an error occurred in the operating systems `close` function, `errno` should be checked for further informations

6.3 Examples

6.4 Implementational Issues

Chapter 7

Class *CEPSet*

7.1 General Description

The basic ideas of class *CEPSet* are presented in the general description of the teamwork library and therefore, we will only repeat some keywords here. The main purpose of an object of class *CEPSet* is to manage a group of connections a process has to others. To identify such a connection uniquely, the *CEPSet* object assigns a **connection id** to it. This id consists of the process id of the peer process and a connection number between 0 and 15. The *CEPSet* object's main task is to provide a comfortable interface to the use of stream connections by matching the connection id's to the CEP's.

Upon creation of an object of class *CEPSet*, it is given the address of its owning process, i. e. the process whichs connections it should manage. The owning process has a so called **acceptCEP** (refer to the masterthesis), and every incoming connection on that CEP is passed to the *CEPSet* object. If a process wants to connect to another process instead of waiting for an incoming connection request, it passes the hostname and the port number of the desired peer's *acceptCEP* to the *CEPSet* object. The *CEPSet* object will then attempt to establish a connection to that process. When a connection was opened successfully, it is entered into the set of connections managed by the *CEPSet* object. Refer to the masterthesis for a more detailed description of how a connection is established.

Besides establishing connections and exchanging data, a *CEPSet* object has a third task, which is to check the administrated connections for special conditions. A special condition, for example, is pending urgent data on a CEP. The *CEPSet* object can be told to look if a so called **exception** has occurred on some of the CEP's and if so, the user can take the appropriate actions to handle the special condition.

Another extension over normal CEP's offered by a *CEPSet* object is the possibility to wait for incoming data on several CEP's simultaneously. The user can specify a group of CEP's (by giving the connection id's) on which to wait for data at the same time. If some data arrives (or is already pending when the respective methods are called) on more then one of the CEP's at the same time, the *CEPSet* object chooses an arbitrary one of them and returns the data read from it. Of course, the user is informed from which CEP the data was read (e. g. to identify the sending process). The methods which allow multiple data sources can be used when the order in which the (possible) senders will transfer their data are not known.

The last thing to mention here, is that the method to accept connections and some of the methods to read data can be given a maximum time interval to wait. If this **timeout** expires before a connection was established or data was read, the methods will abort with an error message. Methods with timeout can be used to detect potential deadlocks because of some sort of data loss or abnormal process termination. An example for the use of these methods can be found in the implementation of class *broadcastBuffer* in section 10.

7.2 Interface

7.2.1 Method CEPSet

Description Constructor for an object of class *CEPSet*. It initializes the attributes and installes the SIGPIPE handler (see below).

Synopsis

```
#include "TWCEPSet.h"

CEPSet( proc *p )
```

Parameters

proc *p a pointer to the owner of the new object

return values

none

Possible Errors

none

7.2.2 Method ~CEPSet

Description Destructor for an object of class *CEPSet*.

Synopsis

```
#include "TWCEPSet.h"

~CEPSet( void )
```

Parameters

none

return values

none

Possible Errors

none

7.2.3 Method accept

Description This method accepts a connection on the owners *acceptCEP*. When a connection can be established, a pointer to the new CEP and the id of the peer process are entered in the list of connections managed by this object.

Synopsis

```
#include "TWCEPSet.h"

ConnId accept( int timeout )
```

Parameters

int timeout optional parameter, if specified, the system will return with an error messsage if no connection was established within the given amount of time.

return values

`INVALID` if an error occurred

the connection id of the new connection if a new connection was accepted

Possible Errors

CEPSet::accept: no accept socket created! the owner of the *CEPSet* object has no valid *acceptCEP*

CEPSet::accept: accept failed(no timeout) the call to `CEP::accept` failed, `errno` should be checked for further informations. This error message can only occur when the *timeout* paramter was not supplied.

CEPSet::accept: accept timed out no connection request arrived during the specified amount of time. The libraries global variable `err` will be set to `TIMEOUT` (defined in `error.h`).

CEPSet::accept: accept failed the call to `CEP::accept` failed, `errno` should be checked for further informations

CEPSet::accept: read failed when the library tried to get the process id of the peer process, the call to `CEP::read` failed. `errno` should be checked for further informations.

CEPSet::accept: no more connections to xx possible the maximum number of connections (`maxConnections`, defined in `const.h`) to the peer process is reached. The accepted connection will not be entered into the set of administrated connections.

7.2.4 Method connect

Description This method tries to connect to the specified process.

Synopsis

```
#include "TWCEPSet.h"

ConnNr connect( procInfo *pi )
```

Parameters

procInfo *pi a pointer to the process information structure (defined in `types.h`) of the process to which a connection should be established

return values

`INVALID` if an error occurred

the connection id of the new connection if a new connection was established

Possible Errors

CEPSet::connect: connect failed the call to `CEP::accept` failed, `errno` should be checked for further informations

CEPSet::connect: write failed when the library tried to send the process id of the owner process to the peer, the call to `CEP::write` failed. `errno` should be checked for further informations

CEPSet::accept: no more connections to xx possible the maximum number of connections (`maxConnections`, defined in `const.h`) to the peer process is reached. The accepted connection will not be entered into the set of administrated connections.

7.2.5 Method read

Description reads the specified amount of data from connection 0 to the process with the given id. The data read is copied into the given memory location.

Synopsis

```
#include "TWCEPSet.h"

int read( IdType from, void* data, int len )
```

Parameters

IdType from the id of the process from which to read data

void* data the memory destination of the data read

int len the amount of data to read

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of data was read successfully

Possible Errors

CEPSet::read: not connected to xx there is no connection to the specified process in the set of connections managed by this object

CEPSet::read: CEP::read failed the call to `CEP::read` failed, `errno` should be checked for further informations

7.2.6 Method readOOB

Description reads the specified amount of out-of-band data from connection 0 to the process with the given id. The data read is copied into the given memory location.

Synopsis

```
#include "TWCEPSet.h"

int readOOB( IdType from, void* data, int len )
```

Parameters

IdType from the id of the process from which to read oob data

void* data the memory destination of the oob data read

int len the amount of oob data to read

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of oob data was read successfully

Possible Errors

CEPSet::readOOB: not connected to xx there is no connection to the specified process in the set of connections managed by this object

CEPSet::readOOB: CEP::readOOB failed the call to `CEP::readOOB` failed, `errno` should be checked for further informations

7.2.7 Method read

Description reads the specified amount of data from a given connection. The data read is copied into the given memory location.

Synopsis

```
#include "TWCEPSet.h"

int read( ConnId c_id, void* data, int len )
```

Parameters

ConnId c_id the id of the connection from which to read data

void* data the memory destination of the data read

int len the amount of data to read

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of data was read successfully

Possible Errors

CEPSet::read: not connected (ConnId: xx) there is no such connection in the set of connections managed by this object

CEPSet::read: CEP::read failed the call to `CEP::read` failed, `errno` should be checked for further informations

7.2.8 Method read

Description Reads a specified amount of data from connection 0 to an arbitrary member of the given group. If the group contains an id of a process, to which no connections exists in the set of connections managed, that id will be removed from the group before the read is done.

Please note, that user can not rely on the order in which the processes of the

group may have sent data to the receiving process. For example, if processes 1 and 2 have sent data to process 3 (process 1 before process 2), it is possible that a (group)read done by 3 returns the data sent by 2 first.
When no data can be read within the specified time (in seconds), this method will return with an error message. If data arrived, it will be written into the specified memory location and the id of the sending process is returned in a result parameter.

Synopsis

```
#include "TWCEPSet.h"

int read( IdGroup* gr, int timeout, IdType* from, void* data, int len )
```

Parameters

IdGroup* gr a pointer to a group of process id's from which data may be sent

int timeout the maximum amount of time to wait for incoming data

IdType* from the result paramter, it will contain the process id of the process which sent the data read

void* data the memory destination of the data read

int len the amount of data to read

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of data was read successfully

Possible Errors

CEPSet::read: not connected to xx, removing from Idgroup there is no connection to a process with this id in the set of connections. The id is removed from the group.

CEPSet::read: select was interrupted by a signal the system call to `select` was interrupted by a signal. The call was restarted.

CEPSet::read: error in select the system call to `select` failed, `errno` should be checked for further informations. The libraries global error variable `err` will be set to `SYSERR` (defined in `error.h`).

CEPSet::read: select timed out no data could be read within the specified amount of time, the libraries global error variable `err` will be set to `TIMEOUT` (defined in `error.h`).

CEPSet::read: read failed the call to `CEP::read` failed, `errno` should be checked for further informations

7.2.9 Method read

Description Reads a specified amount of data from connection 0 to an arbitrary member of the given group. If the group contains an id of a process, to which no connections exists in the set of connections managed, that id will be removed from the group before the read is done.

Please note, that user can not rely on the order in which the processes of the group may have sent data to the receiving process. For example, if processes 1 and 2 have sent data to process 3 (process 1 before process 2), it is possible that a (group)read done by 3 returns the data sent by 2 first.

When no data can be read within the specified time (in seconds), this method will return with an error message. If data arrived, it will be written into the specified memory location and the id of the sending process is returned in a result parameter.

Synopsis

```
#include "TWCEPSet.h"

int read( IdGroup* gr, int timeout_sec, int timeout_usec, IdType* from, void* data, int len )
```

Parameters

IdGroup* gr a pointer to a group of process id's from which data may be sent

int timeout_sec the maximum amount of time to wait for incoming data (seconds)

int timeout_usec the maximum amount of time to wait for incoming data (microseconds)

IdType* from the result paramter, it will contain the process id of the process which sent the data read

void* data the memory destination of the data read

int len the amount of data to read

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of data was read successfully

Possible Errors

CEPSet::read: not connected to xx, removing from Idgroup there is no connection to a process with this id in the set of connections. The id is removed from the group.

CEPSet::read: select was interrupted by a signal the system call to `select` was interrupted by a signal. The call was restarted.

CEPSet::read: error in select the system call to `select` failed, `errno` should be checked for further informations. The libraries global error variable `err` will be set to `SYSERR` (defined in `error.h`).

CEPSet::read: select timed out no data could be read within the specified amount of time, the libraries global error variable `err` will be set to `TIMEOUT` (defined in `error.h`).

CEPSet::read: read failed the call to `CEP::read` failed, `errno` should be checked for further informations

7.2.10 Method read

Description Reads a specified amount of data from an arbitrary connection in the given group. If the group contains an invalid connection id, that id will be removed from the group before the read is done.
Please note, that user can not rely on the order in which the processes of the group may have sent data to the receiving process. For example, if processes 1 and 2 have sent data to process 3 (process 1 before process 2), it is possible that a (group)read done by 3 returns the data sent by 2 first.
When no data can be read within the specified time (in seconds), this method will return with an error message. If data arrived, it will be written into the specified memory location the id of the sending process is returned in a result parameter.

Synopsis

```
#include "TWCEPSet.h"

int read( ConnGroup* gr, int timeout, IdType* from, ConnNr *con, void* data, int len )
```

Parameters

ConnGroup* gr a pointer to a group of connection id's from which data may be read

int timeout the maximum amount of time to wait for incoming data

IdType* from the result paramter, it will contain the process id of the process which sent the data read

ConnNr* con the result paramter, it will contain the connection number of the connection from which the data was read

void* data the memory destination of the data read

int len the amount of data to read

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of data was read successfully

Possible Errors

CEPSet::read: not connected (ConnId: xx), removing from connection group id xx is not valid, it is removed from the group.

CEPSet::read: select was interrupted by a signal the system call to `select` was interrupted by a signal. The call was restarted.

CEPSet::read: error in select the system call to `select` failed, `errno` should be checked for further informations. The libraries global error variable `err` will be set to `SYSERR` (defined in `error.h`).

CEPSet::read: select timed out no data could be read within the specified amount of time, the libraries global error variable `err` will be set to `TIMEOUT` (defined in `error.h`).

CEPSet::read: read failed the call to `CEP::read` failed, `errno` should be checked for further informations

7.2.11 Method read

Description Reads a specified amount of data from connection 0 to an arbitrary member of the given group. If the group contains an id of a process, to which no connections exists in the set of connections managed, that id will be removed from the group before the read is done.
Please note, that user can not rely on the order in which the processes of the group may have sent data to the receiving process. For example, if processes 1 and 2 have sent data to process 3 (process 1 before process 2), it is possible that a (group)read done by 3 returns the data sent by 2 first.

The data read will be copied into the specified memory location and the id of the sending process is returned in a result parameter.

Synopsis

```
#include "TWCEPSet.h"

int read( IdGroup* gr, IdType* from, void* data, int len )
```

Parameters

IdGroup* gr a pointer to a group of process id's from which data may be sent

IdType* from the result paramter, it will contain the process id of the process which sent the data read

void* data the memory destination of the data read

int len the amount of data to read

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of data was read successfully

Possible Errors

CEPSet::read: not connected to xx, removing from Idgroup there is no connection to a process with this id in the set of connections. The id is removed from the group.

CEPSet::read: select was interrupted by a signal the system call to `select` was interrupted by a signal. The call was restarted.

CEPSet::read: error in select the system call to `select` failed, `errno` should be checked for further informations. The libraries global error variable `err` will be set to `SYSERR` (defined in `error.h`).

CEPSet::read: read failed the call to `CEP::read` failed, `errno` should be checked for further informations

7.2.12 Method write

Description Writes a specified amount of data to connection 0 to the process with the given id.

Synopsis

```
#include "TWCEPSet.h"

int write( IdType to, void* data, int len )
```

Parameters

IdType to the id of the process to which the data should be send

void* data the memory location of the data to send

int len the amount of data to send

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of data was written successfully

Possible Errors

CEPSet::write: not connected to xx there is no connection to the specified process in the set of connections managed by this object

CEPSet::write: CEP::write failed the call to `CEP::write` failed, `errno` should be checked for further informations

7.2.13 Method writeOOB

Description Writes a specified amount of out-of-band data to connection 0 to the process with the given id.

Synopsis

```
#include "TWCEPSet.h"

int writeOOB( IdType to, void* data, int len )
```

Parameters

IdType to the id of the process to which the oob data should be send

void* data the memory location of the oob data to send

int len the amount of oob data to send

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of oob data was written successfully

Possible Errors

CEPSet::writeOOB: not connected to xx there is no connection to the specified process in the set of connections managed by this object

CEPSet::writeOOB: CEP::writeOOB failed the call to `CEP::write` failed, `errno` should be checked for further informations

7.2.14 Method write

Description Writes a specified amount of data to a given connection.

Synopsis

```
#include "TWCEPSet.h"

int write( ConnId c_id, void* data, int len )
```

Parameters **ConnId c_id** the id of the connection onto which the data should be written

void* data the memory location of the data to send

int len the amount of data to send

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of data was written successfully

Possible Errors

CEPSet::write: not connected to xx there is no connection with the specified id in the set of connections managed by this object

CEPSet::write: CEP::write failed the call to `CEP::write` failed, `errno` should be checked for further informations

7.2.15 Method write

Description Writes a specified amount of data to connection 0 to the processes in the given group.

Synopsis

```
#include "TWCEPSet.h"

int write( IdGroup* gr, void* data, int len )
```

Parameters

IdGroup* gr a pointer to a group of process id's to which the data should be send

void* data the memory location of the data to send

int len the amount of data to send

return values

`ERROR` if an error occurred

`NOERROR` if the specified amount of data was written successfully

Possible Errors

CEPSet::write: not connected to xx there is no connection to the specified process in the set of connections managed by this object

CEPSet::write: CEP::write to xx failed the call to `CEP::write` to process xx failed, `errno` should be checked for further informations

7.2.16 Method addCEP

Description This method adds the a given CEP as a connection to a certain process. The actually assigned connection number is returned to the calling method.

Synopsis

```
#include "TWCEPSet.h"

ConnId addCEP( IdType id, CEP* cep )
```

Parameters

IdType id The id of the process to which the CEP is connected

CEP* cep the CEP to add

return values

the connection number assigned

`INVALID` if the maximum number of connections to the process with the given id would be execeeded

Possible Errors

CEPSet::addCEP: no more connections to xx possible the maximum number of connections to the process are established

7.2.17 Method getCEP

Description Returns a pointer to the CEP which is associated to the given id and connection number.

Synopsis

```
#include "TWCEPSet.h"

CEP* getCEP( IdType, ConnNr cn )
```

Parameters

IdType id the id of the peer process

ConnNr cn the connection number

return values

a pointer to the associated CEP if the process id/connection number combination is valid

`NULL` if the process id/connection number combination is not valid

Possible Errors

none

7.2.18 Method getCEP

Description Returns a pointer to the CEP which is associated to the given connection id.

Synopsis

```
#include "TWCEPSet.h"

CEP* getCEP( ConnId c_id )
```

Parameters

ConnId c_id the connection id

return values

a pointer to the associated CEP if the connection id is valid

`NULL` if the connection id is not valid

Possible Errors

none

7.2.19 Method isConnected

Description Checks if connection 0 to the process with the given id is established.

Synopsis

```
#include "TWCEPSet.h"

bool isConnected( IdType to )
```

Parameters

IdType to the process id to check

return values

true if connection 0 to process *to* is established

false if connection 0 to process *to* is not established

Possible Errors

none

7.2.20 Method isConnected

Description Checks if a connection with the given id exists.

Synopsis

```
#include "TWCEPSet.h"

bool isConnected( ConnId c_id )
```

Parameters

IdType to the connection id to check

return values

true if the connection exists

false if the connection does not exist

Possible Errors

none

7.2.21 Method exception

Description Waits for a special condition on any of the specified connection. A special condition may be the arrival of out-of-band data. If more then one connection become ready at the same time, the library will return only one chosen arbitrarily.

Please note, that a call to this method blocks until a special condition occurrs. The call will return immediately if a special condition is already pending.

Please note also, that the `select` system call will **not** be restarted if it is interrupted by a signal.

Synopsis

```
#include "TWCEPSet.h"

ConnId exception( ConnGroup *gr )
```

Parameters

ConnGroup *gr a pointer to a group of connection id's on which to wait for an exception

return values

`INVALID` if an error occurred

the connection id of a connection with a special condition if a special condition occurred

Possible Errors

CEPSet::exception: error in select the system call to `select` failed, `errno` should be checked for further informations. The libraries global error variable `err` will be set to `SYSERR` (defined in `error.h`).

7.2.22 Method buildCEPGroup

Description This method builds a group of CEP pointers from a given id group. The CEP's associated with connection 0 of each id group member are placed in the specified CEPgroup.

Synopsis

```
#include "TWCEPSet.h"

void buildCEPGroup( IdGroup* gr, CEPGroup* cgr )
```

Parameters

IdGroup* gr a pointer to a group of process id's whichs CEP's associated with connection 0 will be placed in the result parameter

CEPGroup* cgr a pointer to the resulting CEP pointer group

return values

none

Possible Errors

none

7.2.23 Method buildCEPGroup

Description This method builds a group of CEP pointers from a given connection group. The CEP's associated with the given connection id's are placed in the specified CEPgroup.

Synopsis

```
#include "TWCEPSet.h"

void buildCEPGroup( ConnGroup* gr, CEPGroup* cgr )
```

Parameters

IdGroup* gr a pointer to a group of connection id's whichs associated CEP's will be placed in the result parameter

CEPGroup* cgr a pointer to the resulting CEP pointer group

return values

none

Possible Errors

none

7.2.24 Method assembleConnId

Description This method takes a process id and a connection number and forms a connection id from them.

Synopsis

```
#include "TWCEPSet.h"

ConnId assembleConnId( IdType id, ConnNr cn )
```

Parameters

IdType id the process id

ConnNr cn the connection number

return values

the connection id built from the process id and the connection number

Possible Errors

none

7.2.25 Method resolveConnId

Description This method takes a connection id and breaks it into its parts (a process id and a connection number).

Synopsis

```
#include "TWCEPSet.h"

void resolveConnId( ConnId c_id, IdType* id, ConnNr* cn )
```

Parameters

ConnId c_id the connection id to resolve

IdType* id result parameter for the process id

ConnNr* cn result parameter froo the connection number

return values

none

Possible Errors

none

7.2.26 Method findNextFreeConnNr

Description This method looks for the next free connection number to the process with the given id.

For example, if the local process has already used connections 0, 1 and 2 to process 8, this method will return a value of 3 upon the next call. If the maximum number of connections to a certain process is already installed (`maxConnections`, defined in `const.h`), the call will return an error code.

Synopsis

```
#include "TWCEPSet.h"

ConnNr findNextFreeConnNr( IdType id )
```

Parameters

IdType id the id of the process to which a new connection was established

return values

`ERROR` if the maximum number of connections to the process is exceeded

the next free connection number else

Possible Errors

CEPSet::findNextFreeConnNr: maximum number of connection reached no new connection nnumber can be returned because the maximum of connections to the specified process is already installed

7.2.27 Method getAll

Description Returns a pointer to a group containing the connection id's of all installed connections.

Synopsis

```
#include "TWCEPSet.h"

void getAll( ConnGroup* gr )
```

Parameters

ConnGroup* gr result parameter for the connection id group

return values

none

Possible Errors

none

7.2.28 Method getOwnerId

Description Returns the id of the owner of this object.

Synopsis

```
#include "TWCEPSet.h"

IdType getOwnerId( void )
```

Parameters

none

return values

the id of the owner

Possible Errors

none

7.2.29 Method recvProcessInfo

Description Reads the contents of a process information structure (*procInfo*, defined in `types.h`) sent by the process with the given id.

Synopsis

```
#include "TWCEPSet.h"

int recvProcessInfo( IdType from, procInfo *pi)
```

Parameters

IdType from the id of the process from which to read the process informations

procInfo *pi a pointer to the destination address of the process informations

return values

the result of the CEPSet::read operation

Possible Errors

see `CEPSet::read( IdType, void*, int )`

7.2.30 Method sendProcessInfo

Description Sends the contents of a process information structure (*procInfo*, defined in `types.h`) to the process with the given id.

Synopsis

```
#include "TWCEPSet.h"

int sendProcessInfo( IdType to, procInfo* pi )
```

Parameters

IdType from the id of the process to which the process informations should be send

procInfo *pi a pointer to the source address of the process informations

return values

the result of the CEPSet::write operation

Possible Errors

see `CEPSet::write( IdType, void*, int )`

7.3 Examples

7.4 Implementational Issues

The implementation of class *CEPSet* is not too difficult, but netherless, there are some problems. We will discuss three of them:

- How to perform the matching of connections id's to CEP's.
- How to implement timeouts and how to read from a set of potential senders.
- What happens when a connection breaks unexpectedly?

The first problem seems to be very easy to solve, since we already have objects that were designed to represent associations between objects – the pairsets presented in section 4. The problem with pairsets is, however, that they are designed to match two objects together, but the association which occurs in this context consists of three objects: the process id, the connection number on one side and the CEP on the other. Therefore, we take two steps to do the actual matching. First, the process id and the connection number are packed together into a so called **connection id**. This is done by shifting the process id by four bits to the left and then "oring" it with the connection number. The new value is the connection id which is then matched to the CEP in a *Pairset* object. The *CEPSet* object provides the member function `assembleConnId` and `resovleConnId` to do the folding and unfolding. In most applications, however, where only one connection between two processes is needed, the read/write operations which operate with an implicit connection number 0 will be sufficient.

To answer the second question in the above list, it becomes necessary to take some insight in the UNIX operating system. Normally, any call to a `read` or `accept` function etc., is a so called **blocking call**. This means, that the operating system will not return from that call until some data was read or a connection was established (or a signal occurred, we will return to that later). This functionality is not always useful, because the programmer may want a, e. g., `read` operation to abort if no data was read during a certain amount of time.

Another shortcoming of the operating systems `read` functions is, that they are limited to a single socket. If a process waits for data from multiple other processes and it is not fixed in which order these processes will deliver their data, a fixed sequence of normal read calls will probably cause a deadlock if the data is sent in another then the receiver expected. A more flexible way to read the data – not depending on the order in which the data was sent – is needed.

To solve these two problems, the UNIX operating system offers the `select` system call. The call to this function requires five parameters:

- the highest socket number in use,
- a set of sockets to check for `ready to read`,
- a set of sockets to check for `ready to write`,
- a set of sockets to check for special conditions, and
- a structure to specify the timeout interval.

The call of the `select` function is usually done in four steps. First of all, the user specifies the sockets which he wants to test for either of the three cases (read/write/exception). The operating system provides some macros to deal with the `FD_SET` structure, see the manual pages for `select` for details. If a set is not needed in the `select` call (in most cases, only one of the three sets is used at the same time) a `NULL` pointer must be given to the operating system.

After the socket sets have been specified, the user determines the timeout interval. To do that, he must pass a `timeval` structure (see manual pages for `select`) which matches one of the three conditions to the operating system.

- If he passes a `NULL` pointer to the system, the call to select will be blocked until any of the sockets becomes ready, even infinitely long

- If he passes a `timeval` structure with a value of 0 in the `tv_sec` **and** the `tv_usec` field, the call to `select` will return immediately, i. e. if none of the sockets is ready, the operating system will no wait for one to become ready.
- If he passes a `timeval` structure with a value not equal to 0 in either the `tv_sec` or the `tv_usec` field, the operating system will wait for a socket to become ready (or return instantly if a socket is ready upon the `select` call), but not longer then the time specified in seconds and microseconds.

After the parameters have been set (the first parameter can be set to a default value of `FD_SETSIZE`) the actual call is done. It's behavior is determined by the setting of the last parameter.

After the call returned, the results must be evaluated. The `select` function will return a value of -1 if an error occurred and a value of 0 if the call timed out, i. e. no socket became ready within the specified time. If `select` returns a value > 0, the user must check which of the sockets became ready. The three sets specified in the system call act as so called **value-result parameters**, i. e. they contain informations needed by the operating system when the function call is done, and they contain the results of the operation upon return of the call. To check if a certain socket became ready, the UNIX system provides a macro called `FD_ISSET`, which takes a socket descriptor and a `FD_SET` as parameters. If the socket is in the set, the macro evaluation has a value of 1, else a value of 0. The user must now check one socket after the other to see if it became ready for a certain operation or not.

As we have already mentioned, a `select` call can be interrupted by a signal. This may lead to problems because of premature termination of the call because it is not automatically resumed when the signal handler has finished it's work. To avoid a premature termination because of a signal, our library checks if the call returns because of a signal (`select` returns -1 and `errno` is set to `EINTR`) and if this is the case, the call is repeated until it can finish without being interrupted again.

In a *CEPSet* object, the `select` call is used to implement both, the read/accept methods with timeout and the read methods with multiple sources.

The last question in our list on page 83 was what happens when a connection breaks unexpectedly for any reason. If that happens, e. g. the peer process terminates because of an error, the local process is sent a `SIGPIPE` signal by the Operating system. If this signal is not caught by an appropriate handler, it will cause the process to which it is sent to terminate. To avoid this, we have installed a signal handler which catches the `SIGPIPE` signal. The user will not get aware of the fact that a connection is broken until he attempts to read from or to write to that connection. If he tries to read or write, the teamwork library will issue an error message and the programmer must be prepared to take the appropriate actions to continue with program execution.

Chapter 8

Classes *proc, localProcess, remoteProcess*

8.1 General Description

In this section, we will discuss the objects provided by the teamwork library to represent the processes involved in the distributed application. The reader should therefore be familiar with the UNIX process concept, an introduction to this topic can be found in [Ste90].

We have combined the description of three classes in on section, because their interaction is so close that it would make things more difficult if we tried to present them in isolation. Furthermore, the classes *localProcess* and *remoteProcess* are only renamings of their base class *proc* in order to make the source code of a teamwork application more readable. In other words, the two classes are only introduced for semantic reasons. In the rest of this section, we will first talk about the objects needed to represent local processes (and make clear what we mean with "local" and "remote" processes), and then, we will discuss the objects provided to deal with remote objects.

When a program is started in the UNIX environment, the operating system creates a new process. The user program will the be executed in that process' context. Therefore, the first thing for a user program to do, is to create an object which represents that process. In the teamwork library, this is done by creating a *localProcess* object. After the newly created object is initialized by a call to a member function of class *proc* (the base class of class *localProcess*), the new object will hold informations concerning the process associated with it. These are things such as the name of the host computer where the process is running or the process' pid. During the initialization of the new object, the *acceptCEP* of the associated process is also created. This special CEP is needed by the *CEPSet* objects, refer to section 7 and to the masterthesis. Normally, only one object of class *localProcess* is created in a user program, because that program is running in the same process' context until it terminates.

In most literature on operating systems, no distinction is made between a (user or system) program and it's associated process. Therefore, we will follow this convention, i. e. the terms "program" and "process" will be used interchangeably for the rest of this report.

A distributed application, however, does not consist of a single process, but of a number of processes, probably running on different machines. The program code executed in these processes may be the same or not. It depends on the application programmers choice if he wanted to write one program playing different roles during it's execution or if he wanted to write several programs, one for each role. It is not important which approach to the distributed application is taken. The important point is, that a user program does not need only informations about it's local process, but also about the other processes of the application.

Therefore, we need objects to represent these processes. The *remoteProcess* objects provided by the teamwork library, will contain similar informations about the other processes as for the local process. Furthermore, the creation of a representing object can even be used to **physically create** the process on the remote machine.

To illustrate this, we will now take a look at the starting phase of a teamwork application. The different phases have already been summed up in the masterthesis. Here, we will concentrate on the first two steps (and a bit of step 3) named there.

When a teamwork application is started, the first thing for the (team)**root** to do, is to create an object to represent itself. This is done by creating a *localProcess* object and initializing it. After that, the root process starts the other team members (processes) on the remote machines. Therefore, the root creates a number of *remoteProcess* objects using the `remoteProcess( string hostname, ...)` constructor. These calls will not only create some representing objects for the remote processes, they will also physically create the processes on the host computers. The technical details of the remote process creating are discussed in section 8.6.

The newly created processes will at first create a *localProcess* object to represent themselves. Then, they will call the `remoteProcess( int argc, ...)` to create an object to represent their father process (i. e. the process which started them). The new object which represents the father process will be automatically initialized by the command line arguments. The details of the command line protocol are discussed in section 8.6. These arguments enable the child processes to install a connection to the father process. After a connection is established, the child processes send their process information to the father process which will then call the member function `init()` to initialize the new objects. Please note, that the command line arguments given to a process should not be changed prior to the constructor call for the representing object of the father process. Otherwise, the command line protocol mentioned is likely to fail. The only methods which may be called before constructing the representative for the father process are the methods `isInitial` and `getOutputSpecification`. These methods tell the calling process whether it is the root process of the distributed application and return the output type (described below) that was specified by the creating process, respectively. A call to the second method is only valid when the calling process is not the root process.

To illustrate the start-up phase, we have included the code for this example in section 8.5.

Now, that we have rendered the steps to create remote processes, we will discuss two topics that are interresting in this field:

- how is determined where the output of a remote process should go, and
- how can command line arguments be passed to a newly created process?

To solve the first problem, the `remoteProcess( string hostname, ...)` constructor has two parameters named `outputType` and `outputParam`. At the moment, we have three output types implemented:

XTERM_OUTPUT If this value is passed to the constructor, a new X-terminal window is opened for the new process on the host computer. Any input/output of the new process will then be done via that xterm. The `outputParam` parameter can be used to specify additional informations for the `xterm` command of the UNIX system, e. g. a geometry specification.

FILE_OUTPUT With this value, the output of the new process will be redirected to the file with a name specified by `outputParam`. Please note, that keyboard input is not possible with this output type.

NO_OUTPUT The output of the remote process is redirected to `\dev\null`. As with the **FILE_OUTPUT**, keyboard input is not possible.

Please be aware, that the selection of a certain output type can have a large impact on the execution time of the program. Especially the first output type is extremely costly when the processes have a lot of output. The operating system must not only create an xterm (which implies a number of additional processes), but it must also send any process output to that xterm. This can easily become the most time consuming task of the entire application! Therefore, output type 2 (**FILE_OUTPUT**) should be used if no on-line information about the computation progress is needed.

The second question in the above list was how command line arguments can be passed to the newly created process. Fortunately, this can be done in a very straightforward manner. The `remoteProcess( string hostname, ...)` has a parameter called `params` of type `char*`. Any value specified for this parameter will become the `argv` string for the new process and `argc` will be set accordingly. The second example in section 8.5. illustrates that.

The other member functions of class *proc* do not require any further description since their only task is to manage the access to the internal attributes of a *proc* (or derived) object. The names of the methods should be self explaining.

8.2 Interface class *proc*

8.2.1 Method proc

Description (default) Constructor for an object of class *proc*.

Synopsis

```
#include "TWproc.h"

proc()
```

Parameters

none

return values

none

Possible Errors

none

8.2.2 Method proc

Description Constructor for an object of class *proc*. This constructor will physically create a remote process!

Synopsis

```
#include "TWproc.h"

proc( string remoteHost, outputType output, char *outputParam, IdType yourId, string execfile, char* args, proc *master )
```

Parameters

string remoteHost the name of the host where the new process should be started

outputType output the output type for the new process (see section 8

char* outputParam the parameter for the specified output type

int yourId the id of the new process

string execfile the path of the executable file, relative to the remote accounts home directory

char* args the command line arguments that should be passed to the new process. The values specified here will appear to the new process as if they were passed as arguments from the unix shell.

proc* master a pointer to the process information structure (defined in `types.h`) of the creating process

return values

none

Possible Errors

proc::proc: no such host... a remote host with the specified name does no exist

proc::proc: invalid output type, using xterm the specified output type is not known to the constructor, the default value is used instead

8.2.3 Method proc

Description Constructor for an object of class *proc*. This method is used to create a representing object for the father process. See section 8.6 for details of the command line arguments processed in this constructor.

Synopsis

```
#include "TWproc.h"

proc( int *argc, char **argv, IdType* myId )
```

Parameters

int* argc value/result parameter. On entry, this variable holds the number of command line arguments in `argv`. On exit, it will hold the remaining number of arguments in `argv`, i. e. the number of elements that have not been extracted by the constructor

char argv** a string array of command line arguments

IdType* myid result parameter. On exit, it will hold to id of the child process chosen by the father

return values

none

Possible Errors

none

8.2.4 Method ~proc

Description Destructor for an object of class *proc*.

Synopsis

```
#include "TWproc.h"

~proc( void )
```

Parameters

none

return values

none

Possible Errors

none

8.2.5 Method init

Description Initializes an object of class *proc*.

Synopsis

```
#include "TWproc.h"

void init( procInfo *pi )
```

Parameters

procInfo* pi optional parameter. If specified, the data stored in that structure (defined in `types.h`) will be used to initialize the object. If `pi` is no provided, this method will get the data needed to set the attributes from the operating system and it will also create the *acceptCEP* for the represented process.

return values

none

Possible Errors

none

8.2.6 Method isInitial

Description This method tells the calling process if it is the application's root process or not. Please note, that this method must be called **before** the `proc( int argc, ...)` constructor in order to return a correct result.

Synopsis

```
#include "TWproc.h"

bool isInitial( int argc, char **argv )
```

Parameters

int argc the number of command line arguments in `argv`

char argv** the command line arguments

return values

true if the calling process is the root process

false else

Possible Errors

8.2.7 Method getOutputSpecification

Description Extracts the output type and the output parameters from the command line. Please note, that this method must be called **before** the `proc( int argc, ...)` constructor in order to return a correct result.

Synopsis

```
#include "TWproc.h"

void getOutputSpecification( int argc, char** argv, outputType* output, char* outputParam )
```

Parameters

int argc the number of command line arguments in `argv`

char argv** the command line arguments

outputType* output result parameter for the output type

char* outputParam result parameter for the output parameter

return values

none

Possible Errors

none

8.2.8 Method setId

Description Sets the `id` attribute of this object.

Synopsis

```
#include "TWproc.h"

void setId( IdType val )
```

Parameters

IdType val the value to which the id will be set

return values

none

Possible Errors

none

8.2.9 Method getId

Description Returns the value of the `id` attribute.

Synopsis

```
#include "TWproc.h"

IdType getId( void )
```

Parameters

none

return values

the id of the process represented by this object

Possible Errors

none

8.2.10 Method getPid

Description Returns the value of the tt pid attribute.

Synopsis

```
#include "TWproc.h"

int getPid( void )
```

Parameters

none

return values

the pid of the process represented by this object

Possible Errors

none

8.2.11 Method getHostname

Description Returns a pointer to the *hostname* attribute.

Synopsis

```
#include "TWproc.h"

char* getHostname( void )
```

Parameters

none

return values

a pointer to the name of the host where to represented process resides

Possible Errors

none

8.2.12 Method getPort

Description Returns the value of the `port` attribute.

Synopsis

```
#include "TWproc.h"

int getPort( void )
```

Parameters

none

return values

the port number of the *acceptCEP* of the represented process

Possible Errors

none

8.2.13 Method getProcessInfo

Description Fills the given structure (defined in `types.h`) with the attributes of the represented process.

Synopsis

```
#include "TWproc.h"

void getProcessInfo( procInfo* pi )
```

Parameters

procInfo* pi result parameter for the process informations

return values

none

Possible Errors

none

8.2.14 Method getAcceptCEP

Description Returns a pointer to the *acceptCEP* of the represented process.

Synopsis

```
#include "TWproc.h"

CEP* getAcceptCEP( void )
```

Parameters

none

return values

a pointer to the *acceptCEP*

Possible Errors

none

8.3 Interface class *localProcess*

8.3.1 Method localProcess

Description Constructor for an object of class *localProcess*.

Synopsis

```
#include "TWlProcess.h"

localProcess()
```

Parameters

none

return values

none

Possible Errors

none

8.3.2 Method ~localProcess

Description Destructor for an object of class *localProcess*.

Synopsis

```
#include "TWlProcess.h"

~localProcess()
```

Parameters

none

return values

none

Possible Errors

none

8.4 Interface class *remoteProcess*

8.4.1 Method remoteProcess

Description Constructor for an object of class *remoteProcess*. This constructor will physically create a remote process!

Synopsis

```
#include "TWrProcess.h"

remoteProcess( string remoteHost, outputType output, char *outputParam, IdType yourId, string execfile, char* args, proc *master )
```

Parameters

string remoteHost the name of the host where the new process should be started

outputType output the output type for the new process (see section 8

char* outputParam the parameter for the specified output type

int yourId the id of the new process

string execfile the path of the executable file, relative to the remote accounts home directory

char* args the command line arguments that should be passed to the new process. The values specified here will appear to the new process as if they were passed as arguments from the unix shell.

proc* master a pointer to the process information structure (defined in `types.h`) of the creating process

return values

none

Possible Errors

see `proc::proc( string remoteHost, ...)`

8.4.2 Method remoteProcess

Description Constructor for an object of class *remoteProcess*. This method is used to create a representing object for the father process. See section 8.6 for details of the command line arguments processed in this constructor.

Synopsis

```
#include "TWrProcess.h"

remoteProcess( int *argc, char **argv, IdType* myId )
```

Parameters

int* argc value/result parameter. On entry, this variable holds the number of command line arguments in `argv`. On exit, it will hold the remaining number of arguments in `argv`, i. e. the number of elements that have not been extracted by the constructor

char argv** a string array of command line arguments

IdType* myid result parameter. On exit, it will hold to id of the child process chosen by the father

return values

none

Possible Errors

none

8.4.3 Method ~remoteProcess

Description Destructor for an object of class *remoteProcess*.

Synopsis

```
#include "TWrProcess.h"

~remoteProcess()
```

Parameters

none

return values

none

Possible Errors

none

8.5 Examples

8.6 Implementational Issues

The implementation of most member functions of class *proc* is straightforward and should cause no difficulties when ready the library code. Therefore, the only two methods that we want to discuss a bit more detailed in this section are `proc( string hostname, ...)` and `proc( int argc, ...)` (which are the two `remoteProcess` calls).

The major task of the `proc( string hostname, ...)` constructor is to start a remote process on an arbitrary host by starting a remote shell on that host. Therefore, the `xrsh` UNIX command is used if the output mode is `XTERM_OUTPUT`, else the `rsh` command is used.

To start a remote process with either `xrsh` or `rsh`, some conditions must be fulfilled:

- there must exist user with the same name on the remote machine as the user that started the process which wants to create the remote process. Furthermore, the access rights to the remote account must be set properly (`.rhosts`, etc.).
- the executable file must exist in the location specified by the `execfile` parameter of the constructor. Please note, that the executable's path must be given relative to the home directory of the remote account.

If a library user wants to allow teamwork processes to run under different user id's on the remote hosts, some changes will be necessary to the *proc* constructor. A possible solution would be to add an extra parameter which specifies the user name of the remote account. In this case, the access rights to the remote account must be set by the appropriate UNIX methods (`.rhosts`, etc.).

The second thing to mention in this section is the implementation of the `proc( int argc, ...)` constructor. This method takes the command line arguments passed to the remote process by the operating system. The constructor will at first set some internal variables to the values specified on the command line. The values passed by the father process are:

- the father process' id,
- the port number of the father process' *acceptCEP*,
- the name of the host where the father process resides,
- the pid of the father process,
- the id of the child process (this is a result parameter of the constructor),
- the output type chosen by the father process, and
- the output parameters specified by the father process.

The last two parameters are not assigned to internal variables, their values can be retrieved by a call to `getOutputSpecification` **prior** to the `proc( int argc, ...)` constructor call.

After the internal variables are set, the parameters are deleted from the `argv` string and `argc` is corrected accordingly. After the constructor call is finished, the command line string will look exactly as if the parameters given in the `proc( string hostname, ...)` constructor were passed directly to the new process. By doing this, existing programs that use command line arguments can be integrated in a teamwork library environment without requiring big changes.

Chapter 9

Class *logFile*

9.1 General Description

This class provides a very simple interface to perform logging operations. The user may open an output file, redirect any I/O channels to a file, toggle logging on/off and write data to the logfile as with the 'printf' function. It is also possible to use an existing I/O descriptor as 'logfile', e. g. standard output. The advantage of using this method over the normal screen output lies within the ease of redirection. If, for example, the user has done all screen output by using a logfile with `stdout` as associated descriptor, he only has to insert one function call and all of the former screen output will go a specified file. This can be of great help when debugging programs.

Every logfile has an associated descriptor which may be accessed by derived classes to enhance to basic functionality (refer to the library code for class *errorHandler* for an example). Please note that no check for abuse of the functions (e. g. multiple `open`, `redirect` etc.) is done, so (nearly) the same care as if using some system commands should be taken.

9.2 Interface

9.2.1 Method logFile

Description Constructor for an object of class *logFile*, just initializing some variables.

Synopsis

```
#include "TWlogfile.h"

logFile( void )
```

Parameters

none

return values

none

Possible Errors

none

9.2.2 Method ˜logFile

Description Destructor for an object of class *logFile*, dummy function.

Synopsis

```
#include "TWlogfile.h"

~logFile( void )
```

Parameters

none

return values

none

Possible Errors

none

9.2.3 Method open

Description This method creates a new file or truncates an existing file in order to store any following output into that file.

Synopsis

```
#include "TWlogfile.h"

void open( string fname )
```

Parameters

string fname the name of the output file to use

return values

none

Possible Errors

logFile::open: fopen failed the system call to `fopen` failed, this cause the application to terminate!

9.2.4 Method openNext

Description This method tries to open a file with name `basename<`*i*`>suffix` where *i* is lowest number which is not already in use.

Example: if the current directory contains two files `out1.log` and `out2.log` the procedure call `openNext( "out", ".log" )` will create a file with name `out3.log`

Synopsis

```
#include "TWlogfile.h"

void openNext( string basename, string suffix )
```

Parameters

string basename the first part of the filenames to test for the next filename which is not in use. A natural number will be concatenated to this part which is increased until a free filename is found

string suffix the suffix of the filename group

return values

none

Possible Errors

none

9.2.5 Method open

Description This `open` method sets the associated output descriptor to an already existing descriptor. Of course, given descriptor may be a standard I/O channel, e. g. by using `stdout` the user can do ordinary screen output.

Synopsis

```
#include "TWlogfile.h"

void open( FILE* to )
```

Parameters

FILE* to the address of the descriptor where the output should go

return values

none

Possible Errors

none

9.2.6 Method close

Description Closes the associated descriptor and restores the situation before a redirection if necessary.

Synopsis

```
#include "TWlogfile.h"

void close( void )
```

Parameters

none

return values

none

Possible Errors

logFile::close: fclose failed the system call to `fclose` failed, this cause the application to terminate!

logFile::close: fdopen failed the system call to `fdopen` failed, this cause the application to terminate!

9.2.7 Method redirect

Description This method performs a redirection of output which would normally be written to a specified descriptor. Instead, the output data will be sent to a given file. This function should be called without a previous `open` call.
Please note, that the named file will be destroyed if it already exists.

Synopsis

```
#include "TWlogfile.h"

 void redirect( string fname, FILE* redir )
```

Parameters

none

return values

string fname the name of the output file]

FILE* redir the descriptor which should be redirected to a file

Possible Errors

logFile::redirect: dup failed the system call to `dup` failed, this cause the application to terminate!

logFile::redirect: freopen failed the system call to `freopen` failed, this cause the application to terminate!

9.2.8 Method on

Description Enables output to a specific file or a I/O descriptor. The knows knows where the output was sent before the user called `off` and rebuilds the situation before the `off` call. For example, if the output was sent to a file before `off` was called, the output following the call to `on` will be appended to that file.

Synopsis

```
#include "TWlogfile.h"

void on( void )
```

Parameters

none

return values

none

Possible Errors

logFile::on: dup failed the system call to `dup` failed, this cause the application to terminate!

logFile::on: freopen failed the system call to `freopen` failed, this cause the application to terminate!

9.2.9 Method off

Description Turns logging off. The associated descriptor is closed and an eventual redirection is removed. A subsequent call to `on` will reinstall a previous redirection.

Synopsis

```
#include "TWlogfile.h"

void off( void )
```

Parameters

none

return values

none

Possible Errors

logFile::off: fclose failed the system call to `fclose` failed, this cause the application to terminate!

logFile::off: fdopen failed the system call to `fdopen` failed, this cause the application to terminate!

9.2.10 Method log

Description The given output is written to the associated descriptor (if output is enabled). This method does the same as
`sprintf (<associated descriptor>, format, ... )`
would do. See the manual pages for `sprintf` for a detailed description of the various formats.

Synopsis

```
#include "TWlogfile.h"

void log( string format ... )
```

Parameters

return values

string format the format string (the same as for `printf`)

... a variable number of following arguments of any type.

Possible Errors

logFile::log: no descriptor for logging, exiting ... No output descriptor was opened or assigned on object creation. This will cause the application to terminate.

9.3 Examples

```
void dummy( void )
{
  logFile log;
  log.open( "test.out" );              // write output to "test.out"
```

```
  log.log( "huhu\n" );                // to the output file
  log.off();                          // turn logging off
  log.log( "haha\n" );                // this will never be seen
  cerr << "error0" << endl;           // to stderr
  log.on();                           // turn logging on again
  log.log( "hihi\n" );                // to the output file
  cerr << "error1" << endl;           // to stderr
  log.close();                        // close output file
  log.log( "hehe\n" );                // this will never be seen
  cerr << "error2" << endl;           // to stderr
}

void dummy2( void )
{
  logFile log;
  log.redirect( "test.out", stderr ); // redirect stderr to "test.out"
  log.log( "huhu\n" );                // to the output file
  log.off();                          // turn logging off
  log.log( "haha\n" );                // this will never be seen
  cerr << "error0" << endl;           // to stderr (as before the
                                      // redirection)
  log.on();                           // turn logging on again
  log.log( "hihi\n" );                // to the output file
  cerr << "error1" << endl;           // to the output file
  log.close();                        // close output file
  log.log( "hehe\n" );                // this will never be seen
  cerr << "error2" << endl;           // to stderr (as before the
                                      // redirection)
}
```

The last example shows how to use this class for error logging:

```
void main( void )
{
  logFile error;

  error.open( stderr );
      :
      :
  if( !correct )
```

```
      error.log( "wrong value %d\n", val );   // this will go to stderr
      :
      :
  error.redirect( "err.out", stderr );

  if( wrong )
      error.log( "wrong value %d\n", val2 );  // this will go to "err.out"
      :
      :
  error.close();
}
```

This shows that it is very easy to use the same code for error (file)logging or for error output.

9.4 Implementational Issues

Chapter 10

Class *broadcastBuffer*

10.1 General Description

The purpose of a *broadcastBuffer* object is to provide a transparent, convenient and easy interface to the use of broadcasts in an ethernet/TCP/IP – Unix environment. In the best case, the approach to the use of broadcasts should be the same as to stream sockets. As we will see later, this goal is not reached for 100%, but at least for, say, 95%.

Another important goal during the development of objects for dealing with broadcasts was efficiency. A very basic idea of the teamwork approach is to limit the interprocess communication to few, well defined occasions. During the data exchange phase (**team meeting**), the biggest percentage is normally the transmission of the (large) problem description of the supervisor to the team members. In other words, the simultaneous transfer of (a large amount of) data from one sender to a number of receivers becomes a bottleneck of the teamwork approach. Therefore, we took some effort to make the broadcasting facilities of the teamwork library as efficient as possible. One of the central ideas in this context is to estimate the best data exchange rate by extrapolating transmission information gained in the past (adaptive broadcasting).

Before we discuss the mere technical concerns, we will explain some prerequisites and assumptions about the framework in which the broadcast buffers can be used. As a basic model, we have a number of processes (all runing of different host computers) which can be connected via some sort of secure connection.

One of these processes mentioned has an outstanding position. This position is called the **actual sender**. The sender is the only process which is allowed to send broadcast messages at a certain time, but of course the sender can change at a defined point. Please note, that for each broadcast buffer, there is exactly one sender at a time.

The data exchange (or better, the data transmission from the sender to the receivers) is done in a symetric way. This means, that each send operation for a specific amount of data corresponds to a receive operation for exactly the same amount. If this requirement is violated in an application, it may come to unpredictable results and therefore, the library user should take some care when using broadcasts.

The data transfer between two processes using broadcasts is done in a buffered manner. This means, that a call to a send/receive operation may block because the data buffer is full and has to be flushed (for send) or it is empty and the new contents must be read from the net (for receive). Figure 10.1 shows an example for one of the receivers, the same things apply for the sender in an analogues way.

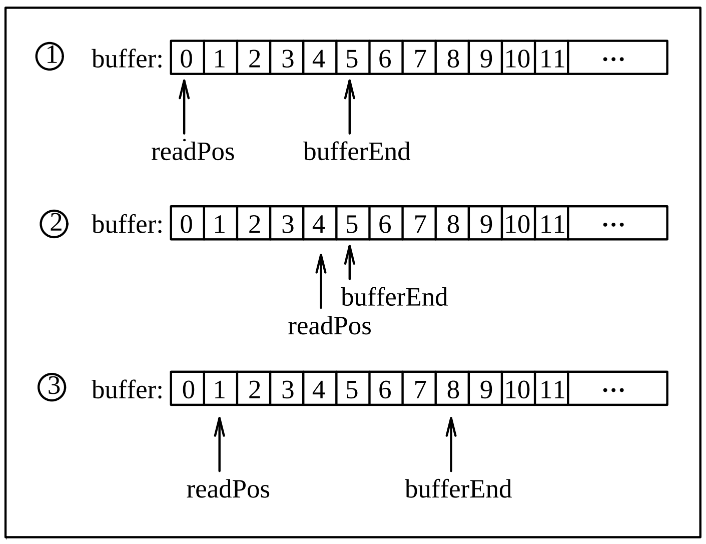

Figure 10.1: Example for a broadcastBuffer read

In the first part of figure 10.1, we have marked the current reading position in the buffer and the end of the buffer (the last position which contains user data). Now, the user calls a read operation for 4 bytes. The request can be fulfilled immediately since the buffer contains enough data, the read pointer is moved 4 bytes further to point to unread data again. If another 4-byte read request arrives, it can not be satisfied immediately because the buffer contains only two bytes (part 2). Therefore, the two bytes are copied to their destination and the broadcast buffer object tries to receive new data from the sender. The users request is blocked until a new data buffer is read, then is completed using the new data (part 3).

As we have said, things are the same for the sender: if a user write request can not be buffered because there is not enough space left, the buffer is broadcasted to the net and then the user request is completed.

By now, it should be clear, why the send/receive operations must be symetric: if the sender and the receivers do not reach their respective buffer-ends at the same time, it may happen that the receivers must wait for incoming data but the sender does not have a filled buffer to flush (or vice versa, i. e. the receivers do not expect to read broadcasts from the net). For the same reason (to prevent receivers from blocking because the sender would not flush a partially filled buffer) it is necessary to end a broadcast phase with an explicit flush operation. It is sufficient to place the flush operation at the end of a broadcasting phase, because the read and write buffers of the sender and the receivers have the same size. In fact, we have only one buffer which is used as read or write buffer since a process can only be sender or receiver – not both at the same time. Because of the equal size, the flushing of a full buffer and the need to get an empty buffer will therefore always fall together (symetric read/write !). Please be aware, that the flush operation is critical because it initiates the full exchange protocol as it will be described later. Therefore, it should be used with care, i. e. only at the end of large transmissions (e. g. the problem description).

Please note, that some overhead will occur if the processes of a broadcast buffer do not reside in the same subnet. This is due to the fact, that each subnet has is own broadcast address. Routers or bridges (see [Tan81]) between two subnets are usually programmed to filter out broadcast messages, i. e. to prevent them from going out of the subnet from which they were sent. It is although possible to sent broadcast messages into a subnet. Therefore, broadcast message to two receivers in different subnets will have to be sent twice: once with the subnet broadcast address of the first receiver, and once with the subnet broadcast address of the second receiver. That means, that the number of transmissions (and thereby the communication latency) grows linearily in the number of subnets involved in a broadcast buffer session.

We will now take a look at the different phases during the lifecycle of a broadcast buffer and explain the most important details of each of the phases. The phases are:

(i) installation of the buffer

(ii) data transfer

(iii) setup changes

(iv) destruction of the buffer

In phase (i), when an object of call *broadcastBuffer* is created, we have to consider two possible situations: (a) the buffer is installed by the initial sender, or (b) it is installed by one of the receivers.

In the first case, the new object must be told three important facts:

- Who is it's owner, i. e. the id of the process it belongs to?
- What set of connections should be used for stream communication?
- Who are the receivers (respectively their id's)?

Upon the call to the constructor, the new object will at first set some internal variables. After that, it will check if the set of connections holds connections to all specified receivers. If some connections are missing, they are installed then. The next thing to do, is to find a common broadcast port on all host computers and then to create a new datagram socket to send (and later possibly receive) broadcasts. Please note, that this is the default behaviour of the *broadcastBuffer* constructor as some of its possible actions can be controlled by some optional parameters (see the reference manual for details)

In the second case, when the new object is created by a receiver, things are quite similar except for the fact, that the new object does not need to know about the group of receivers. After the variables are initialized in the constructor, the new object checks, if it's assinged connection set contains a connection to the initial sender. If not, the object calls `connect` to establish a connection. The sender waits for this *connect* because he does not have a valid connection either (symmetry of connections). After that, the new object receives the common broadcast port and also creates a datagram socket with that port number.

Again, this is only the default behaviour which can be change be using some options.

There are two problems that remain with what we have just said: (a) how to find a common port on all host computers, and (b) how to find the broadcast address, i. e. which address a datagram must have to be handled as broadcast message, of the local network.

The second problem is a merely technical one and therefore, it will be discussed in the "Implementational Issues" part of this manual. The first problem, however, is due to a requirement of the TCP/IP protocol suite. Each broadcast message is addressed to the same port on all computers in the LAN. Therefore, all processes which want to receive a broadcast message for a certain port must open a port with the same number on their respective hosts. We have discussed this problem already in the general description, we will only repeat the protocol to find a common port here. The implementation of the protocol can be found in method `findCommonPort( ... )` in file `broadcastBuffer.cc`.

Step 1 The initial sender tries to open an arbitrary port on it's host. If this works, continue with Step 2, else try again.

Step 2 The sender transmits the port number of the allocated port to the receivers which try to open it on their respective machines.

Step 3 Each receiver sends a message to the sender if the port could have been opened or not.

Step 4 If the port was successfully allocated by every receiver, continue with Step 5, else go to Step 1.

Step 5 The sender and the receivers create a datagram CEP with the common port number. The object becomes the `broadcastCEP` and is used to send or receive broadcast messages.

After an object of class *broadcastBuffer* is created in the initial sender process and in all receivers, broadcasts can be used (phase (ii)). In this second phase, the most important tasks are order keeping and loss detection of the packets sent. As we have already explained in the general description, broadcasts are unreliable datagrams and therefore, they are susceptible to duplication, order confusion or even loss. To illustrate how we handle these problems, we will take a look on a “typical” buffer transmission. Because of the symetric nature of the read and write operations, the sender and the receivers will reach the point when to flush or receive a data buffer at the same time – either because the user requested an explicit flush or due to an internal buffer overflow. For the rest of this report, we will call the transfer of a (even partly filled) buffer a **broadcast session**. A session is subdivided in several **subsessions** which, in terms, consist of a transmission and a acknowledgement/retransmission phase. We will now present the different tasks of the sender and the receivers during these phases.

The first thing for the sender to do, is to check whether all receivers are ready or not. Therefore, the sender waits for a message from every receiver, indicating that it is ready. When all receivers have confirmed to be ready, the data transfer can begin. The sender knows how many bytes have to be sent, and from that value it can calculate the total number of packets needed to flush the entire buffer (#bytes/`packSize`, defined in `const.h`). This number is passed to a method which will estimate the best number of packets to be sent before requesting an acknowledgement. The estimation will lie between a fixed lower bound and the total number of packets needed and it denotes the size of the next subsession. The estimation algorithm is presented in the “Implementational Issues” section of the reference manual.

After that, the sender will send the proposed number of broadcast datagrams. Each of these datagrams has a header and a data field. The header field contains the subsession number (which is **not** reset at the beginning of a new section) and a packet number (which starts from 0 at the beginning of each session – not subsession!). The header contains also an optional operation code to trigger certain actions on the receivers side. The two numbers in the header make it possible, to identify every broadcast message during an entire teamwork application uniquely. The receivers read the datagram messages and the header fields tell them what to do with an incoming packet. Table 10.1 presents the different situations and what has to be done. Please note, that the packet number implies the position into which the contents of data field have to be copied in the buffer (packet number * `packSize`).

The last packet of every subsession contains the header opcode `SendAckn` (or, if it is the last packet of the session, the opcode `EndOfTransmission` – which implies a `SendAckn`). This code tells the receivers to enter the acknowledgement

Situation	Action
(session number == expected) && (packet number == expected)	insert contents of the data field into the buffer (position = `pack.num*packSize`) and mark packet as received
(session number == expected) && (packet number > expected)	insert contents of the data field into the buffer (position = `pack.num*packSize`) and mark packet as received, mark all packets between the received and the expected number as missing
(session number == expected) && (packet number < expected)	insert contents of the data field into the buffer (position = `pack.num*packSize`) and mark packet as received
(session number < expected)	discard the packet

Table 10.1: Possible Actions on Incoming Packets

phase. The packet contain the opcode is protected against loss by the use of a timer. If the sender realizes, that acknowledgements are missing after a certain period of time, it will assume, that at least one receiver did not get the packet containing the acknowledgment request. The last packet will the be repeated until all receivers have sent their acknowledgment or until a maximum number of retransmissions was performed. In the latter case, the protocol will issue an error message and abort because it assumes some irreparable damage at one of the receivers.

The acknowledgment scheme we use is often called a **negative acknowledgment** scheme. This means, that the receivers keep track of which packets the received in some kind of list, and that they tell the sender in their acknowledgement message which packets they did not get. In opposition, a **positive acknowledgement** scheme would be when the receivers report which packets they got and the sender decides which packets must be retransmitted.

However, in our approach, the sender reads the acknowledgement messages of one receiver after the other and sends the missing packets to them using the stream connections. We decided to use stream connections for retransmission for mainly three reasons (refer also to the masterthesis):

- Streams are a secure way of data transmission. Therefore, we do not need an extra protocol to make the retransmission secure.

- The number of lost packets is usually small against the average subsession size. This means that not too many packets are transferred through the (normally) a bit slower streams.
- The packets lost do not necessarily fall together for a number of receivers – a rebroadcasting would gain no advantages.

After all the missing packets have been sent to the receivers, a new subsession begins or the transmission loop ends because the entire buffer was transferred. Figure 10.2 on page 116 illustrates again what we have said in the previous paragraphs.

First of all, the receiver(s) indicate that they are ready to receive. Then, the sender estimates the best subsession size (here: n) and sends the datagrams. The last datagram contains an acknowledgment request opcode. Assuming that the last packet was received correctly (i. e. no timeout occurred), the receivers tell the sender which packets were lost. These packets are retransmitted via stream connections. After the acknowledgement phase is finished, the sender starts to broadcast again, the estimated best subsession size is now m.

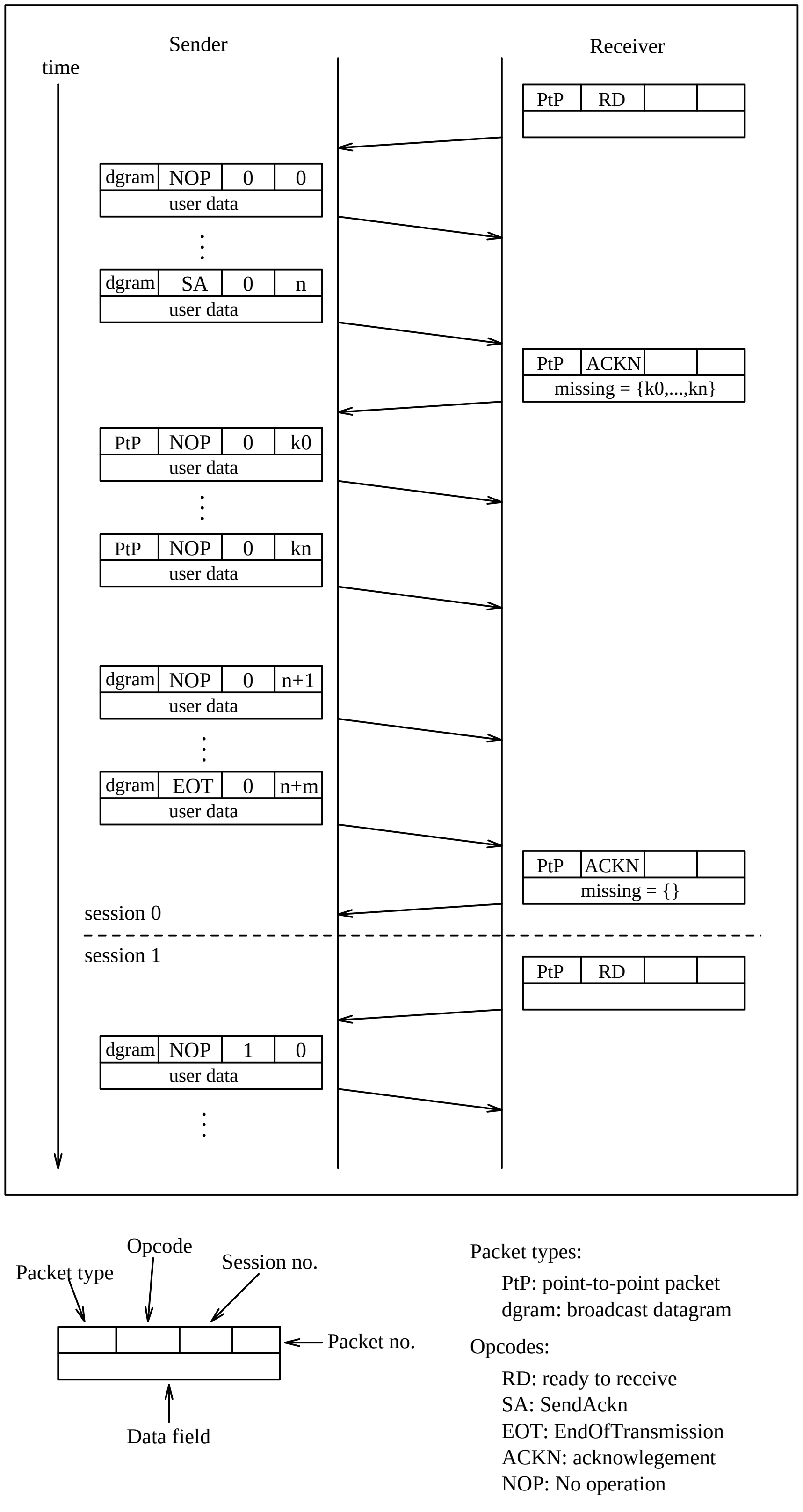

Figure 10.2: Example for a broadcast protocol phase

Again, the last packet contains the acknowledgement request, but this time it is an `EndOfTransmission` opcode, indicating that the entire buffer was transferred. In our example, the receiver did not loose a packet and therefore, no retransmission is necessary. The dashed line should illustrate, that a new buffer transmission phase starts, implying that the session number is increased.

The next important step during the existence of a *broadcastBuffer* object is phase (iii) – the setup change. During this phase, a broadcast buffer can be adopted to new environmental circumstances. This may be things such as sender change (which is, in fact, the only possibility implemented at the moment), buffer size changes etc.

The problem during this stage is, that the receivers do not know which or how may changes will be performed. They can not call the same methods in the same order every time they enter this phase – a more flexible mechanism must be introduced to match the uncertainty of what will happen during phase (iii). The problem is solved by the use of **synchronous events**. A synchronous event can be regarded as a (slightly simplified) remote procedure call (see [Ste90] or [CS93]). The receivers run into a so called **event loop** every time they enter the setup-change phase and wait for incoming events. Every event message contains an operation code which tells the receivers which action to perform, and sometimes also a number of parameters for the opcode. For details of synchronous events, refer to section **??**.

The sender can now send event messages to the receivers, and they will perform the specified method calls locally. When everything is done, the sender uses a message with opcode `QUITLOOP` to end the event loop. The receivers will then resume normal program execution.

As we have already said, the only event which is implemented at the moment, is the sender change event. We will now take a look at how a sender change is realized in detail. First of all, the old sender builds an event message containing the opcode `NEWSENDER` and various informations concerning the process taht should be the next broadcast sender. These informations, include the name of the host where the process resides, or the port number of that process's *acceptCEP*. They are necessary because the receivers of the next broadcast cycle may not be connected to the new sender and so they these informations to open a connection.

This event is then sent to the receivers who will react according to the contents.

The receiver who will be the new sender is sent an extra message by the old sender which contains a list of the id's of the processes which were the receivers in the last cycle. The new sender will then check, if it has a connection to all of the receivers. If not, it will do as many `accept` calls as necessary to establish the missing connections. The receivers which are not the new sender will, in turn, check if they have a connection to the new sender and perform a `connect` call using the data received with the event if necessary.

Please note, that the internal attribute `sender`, which holds the id of the current sender, is not changed until the event loop is left – neither for the old sender nor for any of the receivers. This is important, because the old sender may invoke several other synchronous events and there it may become necessary to check who the actual sender is (which is still the old sender!)

When the old sender wants to end the event loop, it sets the `sender` attribute to it's new value and sends a `QUITLOOP` event to the receivers. When they get the event, they will also commit the changes and a new broadcast cycle can start.

The last step in the lifecycle of an object of class *broadcastBuffer* is it's destruction. This step should occur very seldom, because the installation of a broadcast buffer is costly and therefore, it's lifetime should be reasonably long. Phase (iv) is very short and easy, because no special actions have to be taken. The only two things to do are to print the statistical data collected during the various broadcast sessions (if wanted by the user) and to close the broadcast socket.

These were the different stages during the lifecycle of a broadcast buffer, after studying the examples (and perhaps the code of the teamwork library), the user should have no difficulties with broadcast buffers.

10.2 Interface

10.2.1 Method broadcastBuffer

Description (default) Constructor for an object of class *broadcastBuffer*.

Synopsis

```
#include "TWbrbuffer.h"
```

```
broadcastBuffer( void )
```

Parameters

none

return values

none

Possible Errors

none

10.2.2 Method broadcastBuffer

Description Constructor for an object of class *broadcastBuffer*, normally used by the initial sender. Besides setting the internal attributes, The `options` parameter can be used to ask the new object for additional services. The possible settings (if more then one options has to be used, the values must be combined with the or-operator `|`) are:

NO_OPT No special actions (this is the default)

NO_PORT This option causes the new object to ommit the protocol to find a common port on all host computers of the broadcast buffers in the broadcast group. Please note, that if this option is not set, the creation of the broadcast buffers of the broadcast group must take place synchronously on all host computers. After objects involved in the current broadcast group have found a common port (see method `findCommonPort(...)`, the broadcast socket options are set for the broadcast CEP.

NO_AUTO_CONNECT By default, the new object checks whether a connection exists to every single receiver and if not, it waits for that receiver to connect. This behaviour can be suppressed by specifying this option. Please note, that this will eventually cause the calling process to block if at least one receiver does not have a connection nor does it know that it is supposed to open one.

NO_AUTO_DETECT By default, the new object waits for the receivers to sent their local subnet's broadcast address. Specifying this option suppresses that. Please note, that every broadcast datagram is sent several times over the net if multiple subnets are involved in a broadcast session.

Synopsis

```
#include "TWbrbuffer.h"

broadcastBuffer( IdType  id, CEPSet* conn, IdGroup* recv, brOpt options )
```

Parameters

IdType id the id of the owner process

CEPSet* conn pointer to the stream connections to use

IdGroup* recv the group of receivers associated with this buffer

brOpt options optional parameter which controls the internas of the constructor

return values

none

Possible Errors

broadcastBuffer: tried xx times to find an approriate broadcast port the system could not manage to open a common port on all host computers, the application is terminated.

10.2.3 Method broadcastBuffer

Description Constructor for an object of class *broadcastBuffer*, normally used by the initial receivers. The `options` parameter can be used to ask the new object for additional services. The possible settings (if more then one options has to be used, the values must be combined with the or-operator `|`) are:

NO_OPT No special actions (this is the default)

NO_PORT This option causes the new object to ommit the protocol to find a common port on all host computers of the broadcast buffers in the broadcast group. Please note, that if this option is not set, the creation of the broadcast buffers of the broadcast group must take place synchronously on all host computers. After objects involved in the current broadcast group have found a common port (see method `findCommonPort(...)`, the broadcast socket options are set for the broadcast CEP.

NO_AUTO_CONNECT By default, the new object checks whether a connection exists to the sender and if not, it tries to connect to the sender. This behaviour can be suppressed by specifying this option. Please note, that this will eventually cause the calling process to block if the sender does not have a connection to the receiver and does not exepect to accept one.

NO_AUTO_DETECT By default, the new object sends the broadcast address of the local subnet to the initial sender. Specifying this option suppresses that. As with the **NO_AUTO_CONNECT** option, this option should be used symetrically by the receivers and the initial sender in order to avoid any deadlocks.

Synopsis

```
#include "TWbrbuffer.h"

broadcastBuffer( IdType  id, CEPSet* conn, procInfo* initialSender, brOpt options )
```

Parameters

IdType id the id of the owner process

CEPSet* conn pointer to the stream connections to use

procInfo* initialSender a pointer to a structure which contains the process info of the initial sender, this is needed if a connection has to be established.

brOpt options optional parameter which controls the internas of the constructor

return values

none

Possible Errors

none

10.2.4 Method ~broadcastBuffer

Description Destructor of an object of class *broadcastBuffer*. The broadcast socket and the logfile are closed, if the statistic ouput was enabled, it is now beeing print.

Synopsis

```
#include "TWbrbuffer.h"

~broadcastBuffer( void )
```

Parameters

none

return values

none

Possible Errors

none

10.2.5 Method openPort

Description Opens the broadcast CEP at the given port. If no port number is supplied, he system chooses a port number randomly.

Synopsis

```
#include "TWbrbuffer.h"

openPort( int port = 0 )
```

Parameters

int port the port number for the broadcast port, if no number is specified, the library chooses one randomly.

return values

`ERROR` the port was opened successfully

`NOERROR` the port was not opened for some reason

Possible Errors

none

10.2.6 Method write

Description Writes a specified amount of data into the broadcast buffer. If there is not enough buffer space left, as much as possible is copied into the buffer which is then flushed. Then, the rest of the data is copied into the buffer. This procedure is iterated if necessary, e. g. if the given amount of user data exceeds the buffer size.

Synopsis

```
#include "TWbrbuffer.h"

int write( void* data, int len )
```

Parameters

void* data a pointer to the data to be copied into the broadcast buffer

int len the number of bytes to be written

return values

`ERROR` the data could not be copied into the buffer

`NOERROR` the data was written successfully

Possible Errors

broadcastBuffer::write: error in flush there was not enough buffer space lest, the buffer should have been flushed, but an error occurred during the flush operation

10.2.7 Method read

Description Reads a specified amount of data out of the broadcast buffer and copies it into memory. If there is not enough data in the buffer to fulfill the request, as much as possible is copied into memory and a the method tries to read new buffer from the net. Then, the rest of the data is copied into memory. This procedure is iterated if necessary, e. g. if the specified amount of data exceeds the buffer size.

Synopsis

```
#include "TWbrbuffer.h"

int read( void* data, int len )
```

Parameters

void* data the destination address for the data read

int len the amount of data to read

return values

`ERROR` the data could not be read from the buffer

`NOERROR` the data was read successfully

Possible Errors

broadcastBuffer::read: error in get there was not enough data in the buffer to directly fulfill the request. The method tried to read a new buffer from the net, but an error occurred.

10.2.8 Method flush

Description Flushes the broadcast buffer, i. e. sends its contents as datagrams over the net. See the section 10 for a detailed the protocol description.

Synopsis

```
#include "TWbrbuffer.h"

int  flush( void )
```

Parameters

none

return values

`ERROR` the broadcast buffer could not be flushed for some reason

`NOERROR` the data was sent over the net and was read by all receivers

Possible Errors **broadcastBuffer::flush: not connected to receiver xx** a stream connection (needed e. g. for the acknowledgment packets) to at least one of the receivers is missing. The protocol must be aborted.

broadcastBuffer::flush: receiver(s) not responding at least one of the receivers does not indicated to be ready within `readyTimeOut` (defined in `const.h`) seconds. The protocol is aborted

broadcastBuffer::flush: receiver(s) not acknowledging at least one of the receivers missed the last packet of a subsession which contained the acknowledgement request. If the maximum number of retransmissions of the last packet (`reqAckn`, defined in `const.h`) is not yet reached, the protocol will continue.

broadcastBuffer::flush: retransmitting last packet: xx the last packet of a subsession was lost and is retransmitted.

broadcastBuffer::flush: repeated last message xx times, aborting... the last packet of a subsession was re broadcasted `reqAckn`(defined in `const.h`) times. The protocol is aborted.

10.2.9 Method virtualBroadcast

Description Sets the broadcast buffer transmission to stream connections. i. e. subsequent call to `flush` will use the stream connections to transfer the buffer contents.

Synopsis

```
#include "TWbrbuffer.h"

void virtualBroacast( void )
```

Parameters

none

return values

none

Possible Errors

none

10.2.10 Method physicalBroadcast

Description Sets the broadcast buffer transmission to real broadcasts. i. e. subsequent call to `flush` will use physical broadcast datagrams to transfer the buffer contents. This is the default setting.

Synopsis

```
#include "TWbrbuffer.h"

void physicalBroacast( void )
```

Parameters

none

return values

none

Possible Errors

none

10.2.11 Method get

Description Receives a new buffer from the net, i. e. receives datagrams and copies their contents into the buffer space. See the section 10 for the protocol details.

Synopsis

```
#include "TWbrbuffer.h"

int  get( void )
```

Parameters

none

return values

`ERROR` a new broadcast buffer could not be read for some reason

`NOERROR` a new broadcast buffer was read successfully

Possible Errors

broadcastBuffer::get: unknown command the command field of the last datagram received contained an invalid code

10.2.12 Method changeSetUp

Description This method is only renaming of the eventLoop method from the base class *eventHandler*. It's purpose is to read events and call the handler method `handleEvent` described below. The loop will be left when an event with opcode `QUITLOOP` occurs (but after(!) that event was handled by `handleEvent`).

Synopsis

```
#include "TWbrbuffer.h"

void changeSetUp( void )
```

Parameters

none

return values

none

Possible Errors

none

10.2.13 Method handleEvent

Description The event handler of a broadcast buffer object. Every event read by method `changeSetUp` is passed to this method. If no matching event code is found, the event is passed down to the base class (*eventHandler*) handler.

Synopsis

```
#include "TWbrbuffer.h"

void handleEvent( event* ev )
```

Parameters

event* ev A pointer to a structure containing the event informations (code, paramters etc.). The structure is defined in `types.h`.

return values

none

Possible Errors

none

10.2.14 Method computeConfig

Description This method computes the configuration for the next subsession. 'need' contains the number of packets needed to flush the entire buffer. On return, 'np' will hold the estimated best size for the next subsession and 'sd' will indicate if the send delay should be activated or not. See section 10.4 for details of the estimation algorithm.

Synopsis

```
#include "TWbrbuffer.h"

void computeConfig( int need, int* np, int* sd )
```

Parameters

int need contains the number of packets needed to flush the entire buffer

int* np return value, will hold the estimated best size for the next subsession

int* sd return value, will indicate if the send delay should be activated or not

return values

none

Possible Errors

none

10.2.15 Method findCommonPort

Description This method runs the broadcast buffer initialization protocol to find a common port on all host computers. Refer to section 10 and the general description for the protocol details.

As a side effect of this method, the `broadcast` *CEP* object of the broacast buffer object is instantiated as a datagram socket with the port number found.

Synopsis

```
#include "TWbrbuffer.h"

void findCommonPort( int protocolType, int maxRetry )
```

Parameters

int protocolType This parameter specifies which protocol is actually performed. A value of `SENDER_PROTOCOL` will cause the method to execute the sender side of the protocol, while a value of `RECEIVER_PROTOCOL` will execute the receiver side.

int maxRetry This parameter specifies how many attempts are taken to find a common port before the protocol is aborted.

return values

`ERROR` a common port port could not be allocated after `maxtry` attempts

the port number allocated if a common port was successfully allocated

Possible Errors

broadcastBuffer::findCommonPort: unknown protocol type the protocol type was not valid.

10.2.16 Method getBroadcastAddress

Description Returns the broadcast address of the subnet. See section 10.4 for details.

Synopsis

```
#include "TWbrbuffer.h"

long getBroadcastAddress( t_socket s )
```

Parameters

t_socket s the socket descriptor of the broadcast socket. This value is needed to obtain the list of interfaces that can be used by this socket.

return values

`ERROR` an error occurred

the broadcast address of the subnet no error

Possible Errors

broadcastBuffer::GetBroadcastAddress: gethostname failed the system call to `gethostname` failed, `errno` should be checked for further informations

broadcastBuffer::GetBroadcastAddress: wrong address length the size of the address field in the structure returned by `gethostbyname` is too long

broadcastBuffer::GetBroadcastAddress: wrong netmask the netmask of the subnet is 0, this is not valid

10.2.17 Method addReceiver

Description Add a receiver to the group of receivers of the broadcast buffer. First of all, the new receiver is sent a `SETPORT` event with the broadcast port number currently in use by the buffers of the broadcast group. If it manages to open that port, the receiver is added to the group and the method is done. If, on the other hand, the new receiver is not able to allocate the port, the method issues a `CHANGEPORT` event to all receivers in the broadcast group. The actual sender, the old receivers and the new receiver will then try to find a common port by running the protocol described in sections 10 and the general teamwork overview.

Please note that a connection is not established automatically, this has to be done by the user before the start of a new broadcast session (if a connection between the sender of the next session and the new receiver does not already exist).

Please note also, that this method can only be called by the actual sender, and that it assumes all receivers to be in a `changeSetUp` loop.

Synopsis

```
#include "TWbrbuffer.h"

void addReceiver( IdType new_recv )
```

Parameters

IdType new_recv the id of the new receiver

return values

none

Possible Errors

none

10.2.18 Method removeReceiver

Description Remove a receiver from the group of receivers of the broadcast buffer. No further actions are necessary.

Synopsis

```
#include "TWbrbuffer.h"

void removeReceiver( IdType recv )
```

Parameters

IdType recv the id of the process to be removed from the group of receivers

return values

none

Possible Errors

none

10.2.19 Method setNewSender

Description This method can be called by the actual sender during the setup change phase of a broadcast buffer. It will prepare(!) everything so that the process specified by the paramter will be made the new sender when the event loop is left (and not earlier).

Please note, that this method can only be called by the actual sender, and that it assumes all receivers to be in a `changeSetUp` loop.

Synopsis

```
#include "TWbrbuffer.h"

void setNewSender( procInfo* pi )
```

Parameters

procInfo* pi pointer to a structure containig the process informations of the next sender. The structure is defined in `types.h`.

return values

none

Possible Errors

none

10.2.20 Method getNewSender

Description This method is invoked by the event handler of the broadcast buffer when a `NEWSENDER` event occurs. It will prepare a receiver process for a sender change at the end of the event loop.

Synopsis

```
#include "TWbrbuffer.h"

void getNewSender( char* data )
```

Parameters

char* data a pointer to the event parameters of the `NEWSENDER` event

return values

none

Possible Errors

none

10.2.21 Method setPort

Description This method is invoked by the event handler of the broadcast buffer when a `SETPORT` event occurs. It will instantiates the `broadcast` CEP object as datagram CEP with the given port number and send a message indicating the success (or failure) to the sender of the event.

Synopsis

```
#include "TWbrbuffer.h"

void setPort( event* ev )
```

Parameters

event* ev a pointer to the event structure (defined in `TWtypes.h` event

return values

none

Possible Errors

none

10.2.22 Method changePort

Description This method is invoked by the event handler of the broadcast buffer when a `CHANGEPORT` event occurs. It will close the actual broadcast CEP (`broadcast`) and then try to find a common port by running the protocol described in section 10.

Synopsis

```
#include "TWbrbuffer.h"

void changePort( void )
```

Parameters

none event

return values

none

Possible Errors

none

10.2.23 Method endEvents

Description Ends the event loop and makes the changes prepared during the setup change phase permanet.

Please note, that this method can only be called by the actual sender, and that it assumes all receivers to be in a `changeSetUp` loop.

Synopsis

```
#include "TWbrbuffer.h"

void endEvents( void )
```

Parameters

none

return values

none

Possible Errors

none

10.2.24 Method getLogfileName

Description Returns the filename of the logfile associated with the broadcast buffer.

Synopsis

```
#include "TWbrbuffer.h"

void getLogfileName( string lfn )
```

Parameters

string lfn return value, will hold the name of the associated logfile

return values

none

Possible Errors

none

10.2.25 Method printStat

Description During the destructor execution, the broadcast statistic will be printed to the specified descriptor (which may also be a standart descriptor such as stdout).

Synopsis

```
#include "TWbrbuffer.h"

void printStat( FILE* fp )
```

Parameters

FILE* fp the descriptor for the destination of the statistical data output

return values

none

Possible Errors

none

10.2.26 Method printStat

Description During the destructor execution, the broadcast statistic will be written into the specified file.

Synopsis

```
#include "TWbrbuffer.h"

void printStat( string fn )
```

Parameters

string fn the name of the destination file output

return values

none

Possible Errors

none

10.3 Examples

10.4 Implementational Issues

In this section, we will concentrate on two internal details of an object of class *broadcastBuffer*:

- getting the broadcast address of the LAN
- estimate the best subsession size using statistical data

The first problem occurs, because the TCP/IP protocol suite does not provide a fixed address to mark datagrams as broadcasts. This is due to the fact, that the protocols may be used on different hardware platforms, some of which are capable to do physical broadcasts and some are not. Therefore, we will have to query the (transport) medium access hardware how a data packet can be marked as broadcast. Since we use an ethernet technology as underlying hardware, we know that physical broadcasts are possible and that the only thing to do is to find the correct broadcast address of our LAN.

This is done by a call to the operating system function `ioctl` with parameter `SIOCGIFCONF`. This call returns a list containing informations about all the (hardware) medium access interfaces of the local host. For our purposes, it does almost not matter which of the interfaces we choose, since any of the contains the needed information. As anywhere, this rule has an exception and this is the so called **loopback** interface. This hardware interface mirrors any data put on the ethernet back to the local host. It is needed to do the medium access collision detection of the CSMA/CD protocol. For details of the this hardware access protocol, the reader should see [Tan81].

The other interfaces, however, contain the information we want and this is the **netmask** of the local area network. Together with a TCP/IP address of the local host, the broadcast address of the LAN can be computed as follows:
`broacastAddress = (host_adr & netmask) | (0xffffffff & ~netmask)`

By definition, any packet which has this value of as destination address is considered to be a broadcast message.

The interface list is searched for an interface which is not the loopback interface and the broadcast address is computed according to the formula above and returned by method `getBroadcastAddress`.

Problems may occur, if the name of the loopback interface of the LAN where the teamwork library should be used is different from the definition in `const.h`. Therefore, if the system does not send broadcast messages, the user should check if the name of the loopback interface as reported by the Unix command `ifconfig -a` is the same as defined in `brbuffer.h`. If not, the definition should be adopted accordingly.

The next thing we want to discuss is our estimation algorithm for the size of the next subsession. As we have already said, we use several statistical data collected in earlier subsessions to estimate the next value. The data we use are:

- the average packet loss during the last `averageLossHistory` (defined in `const.h`) subsessions. To obtain this value, we compute a weighted arithmetic mean of the packet losses of the last `averageLossHistory` subsessions. We use weighted mean, because we think that the losses recorded in earlier subsessions should be taken less into account then more recent data. Therefore, our formula to calculate the mean is:

$$avLoss = \frac{\sum_{i=act-averageLossHistory}^{act-1}\left((\frac{1}{2})^{act-i-1} \cdot \sum_{j=0}^{numRecv}(loss_j(i))\right)}{\sum_{k=0}^{averageLossHistory-1}(\frac{1}{2})^k} \tag{10.1}$$

  In equation 10.1, act is the number of the actual subsession, $numRecv$ holds the number of receivers and the function $loss_j(i)$ returns the number of packets lost by receiver j in subsession i. Please note, that the sum is not evaluated if i is less then 0. Please note also, that if the divisor of equation 10.1 is 0, $avLoss$ is assigned a value of `averageLossDefault`, defined in `const.h`.

- the absolute number of lost packets containing opcode `SendAckn` or `EndOfTransmission` during the last `acknReqLossHistory` subsessions. This value is critical for broadcast performance, because every lost acknowledgement request packet implies a delay (to detect that it was lost) and a retransmission. Therefore, special activities should be taken if this value exceeds a certain bound. The absolute number is computed by:

$$absReqLoss = \sum_{i=act-acknReqLossHistory}^{act-1} \sum_{j=0}^{numRecv} acknReqLoss_j(i) \quad (10.2)$$

  Again, act holds the number of the actual subsection, $acknReqLoss_j(i)$ returns the number of packets lost by receiver j in subsession i. As in in equation 10.1, the sum is not evaluated if i is less then 0.

- the average size of the last `avSSSHistory` subsessions. Again, we use a weighted arithmetic mean to reduce the influence of older data.

$$avSSS = \frac{\sum_{i=act-avSSSHistory}^{act-1}\left(\left(\frac{1}{2}\right)^{act-i-1} \cdot sss(i)\right)}{\sum_{0}^{avSSSHistory-1}\left(\frac{1}{2}\right)^{k}} \quad (10.3)$$

  In equation 10.3, $sss(i)$ returns the size of subsession i. If i does not denote a valid subsession number (e. g. if $act - avSSSHistory < 0$), the return value is ignored in the computation.

Now we will explain, how we use these values to set certain variables which will affect the behavior of a broadcast buffer during the sending phase. First of all, the size of the next subsession is chosen according to:

$$nextSSS = \begin{cases} lastSSS + increaseUnitSize & if \quad avLoss \leq littleLoss \\ avSSS & if \quad littleLoss < avLoss \\ & \quad\ \leq mediumLoss \\ avSSS - avLoss & if \quad mediumLoss < avLoss \\ & \quad\ \leq highLoss \\ avSSS - avLoss - decreaseUnitSize & if \quad avLoss \geq highLoss \end{cases} \quad (10.4)$$

Here, $lastSSS$ refers to the size of the last subsession. The values for `littleLoss, mediumLoss, highLoss, increaseUnitSize` and `descreaseUnitSize` are defined in `const.h`

Another fact which influences the senders behavior is whether the **send delay** is activated or not. If this value is set, the last packet of every subsession (containing the acknowledgement request) is sent twice, with a little delay (`sendDelay` microseconds) between them. This helps to reduce the number of acknowledge request losses because the delay enables the receivers to process some datagrams received earlier thus freeing internal buffer space for more incoming packets. The delay option is turned on if the number of acknowledge request packests lost exceeds the value of `absAcknReqLossBound`, defined in `const.h`.

Please note, that the first call to method `computeConfig` is a rather special case, because there is no statistical data about earlier phases available at that time. Therefore, the size of the next (in fact, the first) subsession is chosen to be $\frac{1}{4}$ of the number of packets needed to send the entire buffer. This value seems to be chosen a bit arbitrarily – and it is.

Chapter 11

Class *brStatistic*

11.1 General Description

The purpose of an object of class *brStatistic* is to hold detailed informations about the last `brStatSize` (defined in `const.h`) broadcast subsessions during which the process which created the statistic object was sender. If the number of broadcast sessions during which that process was sender in a teamwork application exceeds this limit, the data about earlier phases are forgotten (unless the user takes explicit actions to save these data).

In the teamwork library, the collected data is used for two purposes:

- The on-line estimation of the best size for the next subsession and to decide whether to activate a send delay for the last packet of each subsession.
- The off-line analysis of the broadcast traffic. A visualization of the collected data may help to customize the protocol to site specific needs, and thereby improve the protocol's efficiency.

There are two types of data which are collected during a broadcast subsession:

- The receiver specific information (`recvTransmissionInfo`, RTI) contains for every broadcast subsession and every receiver involved:
  - the packet loss, and
  - the acknowledgement request loss.

- The general transmission information (`genTransmissionInfo`, GTI) holds the informations about a subsession which are receiver independent:
  - the number of bytes transferred,
  - the number of packets sent,
  - the information if a send delay was activated or not, and
  - the weighted arithmetic mean of the packet losses over the last `averageLossHistory` (defined in `const.h`) subsessions.

The data about the subsessions is recorded using the methods of the *brStatistic* object, but it lies in the responsibility of the programmer to supply the object with the data to store. In this context, it is very important to mention, that the broadcast number used to access the data from earlier broadcast session is *not* the same number as the subsession number used from the objects of class *broadcastBuffer* to identify a certain subsession. The number used to access earlier broadcast related data from objects of this class refers to subsessions in which the calling process was sender. If a library user wants to extend the broadcast buffers supplied, care should be taken that the statistic is kept up to date. The average library user, however, should not be concerned with this problem.

In the appendix of the general description, we present a little tool we have developed to visualize some of the statistical data. This tool can be used to "tune" some of the broadcast buffers settings.

11.2 Interface

11.2.1 Method brStatistic

Description Constructor for an object of class *brStatistic*

Synopsis

```
#include "TWbrstat.h"

brStatistic( void )
```

Parameters

none

return values

none

Possible Errors

none

11.2.2 Method ~brStatistic

Description Destructor for an object of class *brStatistic*.

Synopsis

```
#include "TWbrstat.h"

~brStatistic( void )
```

Parameters

none

return values

none

Possible Errors

none

11.2.3 Method setReceivers

Description Tells the *brStatistic* object which receivers participate in the next subsession. Only statistical data of announced processes will be recorded.

Synopsis

```
#include "TWbrstat.h"

void setReceivers( IdGroup* recv )
```

Parameters

IdGroup* recv a pointer to the id's of the participating receivers

return values

none

Possible Errors

none

11.2.4 Method incNum

Description Increments the current write position in the data array. If the new position exceeds the maximum array size, new incoming data will be written to the beginning of the array again (sliding window technique, see section 11.4).

If an overflow occurs and the user has enabled a print-on-overflow (by a prior call to method `printOnOverflow`, the `Print` method of the statistic object is called. That method will print the statistical data of the last `brStatSize` (defined in `const.h`) broadcast subsessions.

Synopsis

```
#include "TWbrstat.h"

int incNum( void )
```

Parameters

none

return values

1 an overflow occurred

0 no overflow

Possible Errors

none

11.2.5 Method getNum

Description Returns the actual subsession number used by the statistic object.

Synopsis

```
#include "TWbrstat.h"

int getNum( void )
```

Parameters

none

return values

the actual subsession number

Possible Errors

none

11.2.6 Method n_Packs

Description Sets the number of packets of the current subsession to a given value.

Synopsis

```
#include "TWbrstat.h"

void n_Packs( int val )
```

Parameters

int val the number of packets of the current subsession

return values

none

Possible Errors

none

11.2.7 Method incBytesSent

Description Increments the number of bytes sent during the current subsession by a given value.

Synopsis

```
#include "TWbrstat.h"

void incBytesSent( int val )
```

Parameters

int val the number of bytes to add

return values

none

Possible Errors

none

11.2.8 Method send_delay

Description Records whether the send delay is activated during the current subsession or not.

Synopsis

```
#include "TWbrstat.h"

void send_delay( int val )
```

Parameters

int val if `val` is greater then 0, the send delay will be recorded as activated, else as not activated

return values

none

Possible Errors

none

11.2.9 Method avLoss

Description Sets the weighted arithmetic mean of the packet losses over the last `averageLossHistory` (defined in `const.h`) subsessions to the given value.

Synopsis

```
#include "TWbrstat.h"

void avLoss( double val )
```

Parameters

double val the weighted arithmetic mean

return values

none

Possible Errors

none

11.2.10 Method get_n_Packs

Description Returns the number of packets sent during the specified subsession.

Synopsis

```
#include "TWbrstat.h"

int get_n_Packs( int num )
```

Parameters

int num the subsession of interrest

return values

`ERROR` an error occurred

the number of packets sent the subsession number was ok

Possible Errors

brStatistic::get_n_packs: invalid subsession number xx the subsession number was not in the valid range

11.2.11 Method wasPadding

Description Returns true if the specified subsession was a padding session, else returns false.

Synopsis

```
#include "TWbrstat.h"

bool wasPadding( int num )
```

Parameters

int num the subsession of interrest

return values

`true` if the specified subsession was a padding session

`false` if the specified subsession was not a padding session

Possible Errors

brStatistic::wasPadding: invalid subsession number xx the subsession number was not in the valid range

11.2.12 Method get_bytes_sent

Description Returns the number of bytes sent during the specified subsession.

Synopsis

```
#include "TWbrstat.h"

int get_bytes_sent( int num )
```

Parameters

int num the subsession of interrest

return values

`ERROR` an error occurred

the number of bytes sent the subsession number was ok

Possible Errors

brStatistic::get_bytes_sent: invalid subsession number xx the subsession number was not in the valid range

11.2.13 Method get_send_delay

Description Checks whether send delay was activated during the specified subsession or not.

Synopsis

```
#include "TWbrstat.h"

int get_send_delay( int num )
```

Parameters

int num the subsession of interrest

return values

`ERROR` an error occurred

1 the subsession number was ok and send delay was activated during that subsession

0 the subsession number was ok and send delay was not activated during that subsession

Possible Errors

brStatistic::get_send_delay: invalid subsession number xx the subsession number was not in the valid range

11.2.14 Method PacketLost

Description Records a packet loss of the specified receiver during the actual subsession.

Synopsis

```
#include "TWbrstat.h"

void PacketLost( IdType who )
```

Parameters

IdType who the id of the receiver who lost a packet

return values

none

Possible Errors

none

11.2.15 Method AcknReqLost

Description Records a acknowledgement request loss of the receivers in the specified group during the actual subsession.

Synopsis

```
#include "TWbrstat.h"

void AcknReqLost( IdGroup* who )
```

Parameters

IdGroup* who pointer to the group of id's of the receivers who lost an acknowledgement request

return values

none

Possible Errors

none

11.2.16 Method GetPacketLoss

Description Returns the packet loss of the specified receiver during the given subsession.

Synopsis

```
#include "TWbrstat.h"

int GetPacketLoss( int num, IdType who )
```

Parameters

int num the subsession of interrest

IdType who the receiver who's packet loss should be returned

return values

`ERROR` an error occurred

the packet loss of the receiver no error

Possible Errors

brStatistic::GetPacketLoss: invalid id the given id was not valid, i. e. a receiver with that id was not recorded during the specified subsession

brStatistic::GetPacketLoss: invalid subsession number xx the subsession number was not in the valid range

11.2.17 Method GetAcknReqLoss

Description Returns the acknowledgement request loss of the specified receiver during the given subsession.

Synopsis

```
#include "TWbrstat.h"

int GetAcknReqLoss( int num, IdType who )
```

Parameters

int num the subsession of interrest

IdType who the receiver who's acknowledgement request loss should be returned

return values

`ERROR` an error occurred

the acknowledgement request loss of the receiver no error

Possible Errors

brStatistic::GetAcknReqLoss: invalid id the given id was not valid, i. e. a receiver with that id was not recorded during the specified subsession

brStatistic::GetAcknReqLoss: invalid subsession number xx the subsession number was not in the valid range

11.2.18 Method printOnOverflow

Description Sets the class attribute `poo` to the given value. If `poo` is set when an overflow occurs during a call to method `incNum`, the last `brStatSize` (defined in `const.h` broadcast statistics are printed.

Synopsis

```
#include "TWbrstat.h"

void printOnOverflow( bool p )
```

Parameters

bool p the value to assign to `poo`, default is `true`

return values

none

Possible Errors

none

11.2.19 Method Print

Description Prints the statistical data currently in the array.

Synopsis

```
#include "TWbrstat.h"

int Print( void )
```

Parameters

none

return values

`ERROR` an error occured

`NOERROR` the statistic was printed

Possible Errors

brstat::Print: no descriptor specified the logfile associated with the statistic objecy was not opened properly

11.3 Examples

11.4 Implementational Issues

The main problem with an object of class *brStatistic* is, that we a potentially infinitely long sequence of data packets (each containing informations about one broadcast subsession) which we do not want to save entirely. We would like to hold a fraction of that stream, consisting of data about the most recent subsessions.

If we assume the data packets to be successively numbered starting with 0, and if we assume further, that we have a fixed amount of memory cells (numbered from 0 to n) to hold a fraction of the stream, our problem is the following: if k is the highest packet number used at the moment, we have to find a possibility to hold the packets from $k-n$ to k in our array for every k. If $k-n$ is less then 0, only the packets from 0 to k have to be stored.

The easiest way to solve this problem, is to add an extra field to every data packet which holds its number. The problem with this idea is, that the entire list must be searched for any packet access. This is very costly, and therefore, we decided to use another approach. Our solution is to use a storing technique which is know as **sliding window**. A sliding window can be considered as a special form of an array (in fact, arrays are used in most cases to implement sliding windows), the main difference lies within overflow handling. With the use of sliding windows, we have two outstanding positions: the actual window **start** position, and the the current **write** position. The write position is incremented on every new data packet and then the data is written into that position. If it reaches the array end, it starts again at position 0 and the window start pointer is increased by one position. This pointer will also start from 0 when the array end is reached. And here can see the difference to ordinary arrays: a sliding window is a kind of a cyclic structure which can hold a fixed amount of data. When the (ring)array is full, new incoming data overwrites the oldest data in the ring.

Figure 11.1 illustrates the behavior of a sliding window when it is full. The numbers **in** the boxes represent the data packets stored in the window, whereas the numbers **over** the boxes stand for the physical array indexes. In the first part of the figure, we see that the write pointer (wp) points to the last array cell.

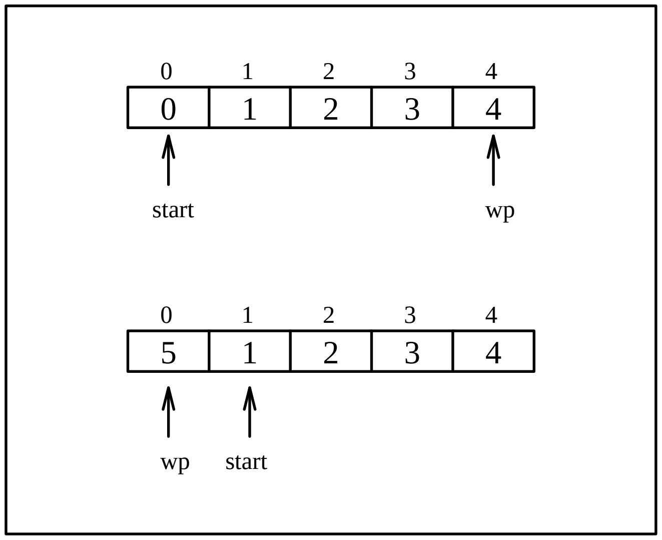

Figure 11.1: Example for a sliding window access

Now, if the next data packet arrives, the write pointer is set to the first array cell again, the data is written into that position and the window start moves one position further. This process is iterated on every new incoming packet, i. e. the array holds the last `arraySize` packets of an incoming packet stream.

Now, what is the advantage in using a sliding window over the use of a normal array? The main advantage is, that any copy operations (which would be necessary to free an array cell if the array is full) can be omitted. The elements in a sliding window can be accessed directly by performing a simple index calculation (packet number modulo the array size) which matches the packet numbers directly to the (physical) array positions. Therefore, this method is a good compromise between efficiency (access speed) and economy (memory usage).

The last problem that remains with the use of sliding window arrays, is what to do with the data that will be overwritten if the window is full and new data arrives. One possibility would be to silently overwrite the data without informing the user. This is a very rude solution and it is very unsatisfying for the user if he wants to save the old data. Therefore, our *brStatistic* objects inform the user every time when as much data packets have been stored as the array has cells. This check is very easy, since it is always the case when `br_num brStatSize == 0`(see the implementation of `incNum`).

When the user saves the old data at these well defined points, he need not worry about duplicate data since the elements do not overlap. The *brStatistic* objects already have built in a method to save the older data. When an overflow occurs, the old data is written to the associated logfile (if the user enabled this by performing a `printOnOverflow` call).

Chapter 12

Classes *asynEventHandler* and *userHandler*

12.1 General Description

In a distributed application, it sometimes becomes necessary that processes communicate outside of the well defined points in the normal program execution. Examples for this need are unexpected error situations or cases where one process has urgent messages for the other. In a teamwork application, e. g., the entire application can be terminated when one of the team members has found the solution. In such a case, it is not necessary to wait until the next team meeting. To provide an interface to this so called **asynchronous communication** (i. e. outside of the normal flow of control), we have already introduced our concept of **asynchronous events** (refer to the masterthesis for details). We will now take a closer look at the usage of those events.

If one process sends an asynchronous event to another, the teamwork library needs to take two steps to inform the receiving process about the occurrence of an that event. First of all, the teamwork library must be informed when an asynchronous event is sent to a process. This is done by the operating system, which detects the arrival of an urgent message. (see section 12.4). The urgent data features of the TCP/IP protocol suite were used to implement the asynchronous events. The operating system will then send a signal to the process to which the event was sent and the libraries signal handler will be invoked. In the second step of the asynchronous event delivery – after some checks – a **user**

handler is called to handle the event.

If we look at the previous paragraph, the first two steps which have to be done by a library user who wants to work with asynchronous events become obvious. First, the user must create the asynchronous event handler object of the library, then he must supply a user handler and announce it to the libraries handler. The user handler will be invoked on every asynchronous event.

When the libraries asynchronous event handler is created, it gets a pointer to a *CEPSet* object as parameter. That set of connections will be used to send the asynchronous events to the receivers. The term "handler" is therefore a bit misleading for objects of this class, since they do not only handle incoming events, but they have also methods to send events to other processes. The second parameter to the constructor of an `asynEventHandler` object is a pointer to a **user event handler**, i. e. the portion of user code which is executed when and asynchronous event arrives. If this parameter is omitted, a default handler will be used.

Now, let us take a look at how a user handler is implemented. An event handler for asynchronous events designed by a library user must be derived from class *userHandler*. This class has only one member function (`processEvents`), which is called on the occurrence of an asynchronous event. Therefore, the library user should implement a class which is derived from *userHandler* and override the default handler function with a function to his like. This may all sound a bit confusing, but a look at the examples in section 12.3 will explain how a library handler is installed and how a user handler is created.

The next thing to mention in the context of asynchronous events, are possible restrictions in their use. Sometimes, a process may want to proceed in its program execution without being interrupted. Therefore, we will introduce a concept which allows a process to limit the group of processes who are allowed to send asynchronous events to it. To determine a group of possible senders, an object of class `asynEventHandler` has two member functions `ignoreAsynEvents` and `allowAsynEvents`. Each of the the methods takes either a single process id or a pointer to a process group as parameters. If a process is in the group of processes from which asynchronous events are ignored, the library handler will catch the event but it will not call the user handler. An event from such a process is simply discarded.

The concept of ignoring events is very useful, but it also has some disadvantages. The biggest disadvantage is, that the events are discarded and not stored for later delivery. If an event is ignored, the sending process does not know that the event was not delivered to the receiver and this may lead to error situations. Therefore, it would be good to have a possibility to **block** asynchronous events for a certain time. This can be achieved with the member function `blockAsynEvents`. Any asynchronous events which occur after this method was called are suspended until the user calls `unblockAsynEvents`. After that, the events are delivered instantly. The advantage of this method is, that no events are lost. The disadvantage, on the other hand, is, that the blocking can not be limited to certain possible senders. The UNIX operating system allows either all or no asynchronous events to be blocked.

There is one very important limitation in the use of asynchronous events which should be always kept in mind when using them: when one asynchronous is handled, it can **not** be interrupted by another asynchronous event. As a consequence of this limitation, the user handler should be very short and it should not perform any complex operations. Typically, a user handler will only set some (global) variables to communicate with the normal flow of operation.

Please note also, that only one *asynEventHandler* object can be created. This lies within the fact, that the UNIX operating system does not allow more then one handler for a signal (here the `SIGURG` signal). It is, however, possible to have more then one user handler, although only one of them can be active at a time. If the user has implemented several handlers, one of them can be activated with the member function `setUserHandler` of class *asynEventHandler*.

Before we present the user interface to objects of class *asynEventHandler* in the next section, we have to mention a requirement of the operating system concerning the use of urgent data, which are used to implement the asynchronous events of our library. Before this type of data can be received on a socket, that socket must be marked as "asynchronous". To mark all connections in the *CEPSet* given as constructor argument as asynchronous, the user must call the member function `enableAsynEvents`. Normally, this is done right after the creation of the *asynEvntHandler* object. When a new connection is added to the *CEPSet* object, it is **not** marked as asynchronous automatically. Therefore, the user has to call the member function `addAsynSender` for every connection that is established after the call to `enableAsynEvents`.

12.2 Interface

12.2.1 Method asynEventHandler

Description Contructor for an object of class *asynEventHandler*.

Synopsis

```
#include "TWasynEventHandler.h"

asynEventHandler( CEPSet* conn, userHandler* ueh )
```

Parameters

CEPSet* conn pointer to the set of connections to use to send and/or receive asynchronous events

userHandler* ueh optional Parameter, if it is supplied, it tells the this object which user handler to call on the occurrence of an asynchronous event. If no user handler is specified, the default handler `defaultAUEH` will be installed.

return values

none

Possible Errors

asynEventHandler::asynEventHandler: asynEventHandler already exists the library user tried to create a second *asynEventHandler* object. This is not allowed.

12.2.2 Method ~asynEventHandler

Description Destructor for an object of class *asynEventHandler*. The signal handler for urgent data (`SIGURG`) is reset to it's default value and the asynchronous event handler is marked as `not installed`.

Synopsis

```
#include "TWasynEventHandler.h"

~asynEventHandler( void )
```

Parameters

none

return values

none

Possible Errors

none

12.2.3 Method setUserEventHandler

Description Sets the user handler to the given object. The new object must be of a class derived from class *userHandler*. Upon the occurrence of an asynchronous event, method `processEvent` of the specified object is called.

Synopsis

```
#include "TWasynEventHandler.h"

void setUserEventHandler( userHandler* ueh )
```

Parameters

userHandler* ueh a pointer to the user handler

return values

none

Possible Errors

none

12.2.4 Method enableAsynEvents

Description This method must be called before any asynchronous messages can be sent to a process. It will install the library handler for urgent data and mark the sockets administrated by the *CEPSet* object specified in the constructor call as **asynchron**, i. e. the sockets are enabled to receive asynchronous messages.

Synopsis

```
#include "TWasynEventHandler.h"

int enableAsynEvents( void )
```

Parameters

none

return values

`ERROR` an error occurred

`NOERROR` the library handler was installed and the sockets are marked as asynchron

Possible Errors

asynEventHandler::enableAsynEvents: no CEP the call to `CEPSet::getCEP` returned a `NULL` value

asynEventHandler::enableAsynEvents: fcntl:setown failed the system call to `fcntl` with command `setown` failed, `errno` should be checked for further informations

asynEventHandler::enableAsynEvents: fcntl:fasyn failed the system call to `fcntl` with command `fasyn` failed, `errno` should be checked for further informations

12.2.5 Method addAsynSender

Description This method must be called when a new connection is added to the *CEPSet* used by the asynchronous event handler to explicitly mark the associated socket as asynchron. If the second parameter is omitted, the method will use connection 0 to the process with the specified id.

Synopsis

```
#include "TWasynEventHandler.h"

int addAsynSender( IdType possibleSender, ConnNr cn )
```

Parameters

IdType possibleSender the id of the process which may sent asynchronous events

ConnNr cn optional parameter, if specified, it denotes the connection to the process with id `possibleSender` that should be marked as asynchron

return values

`ERROR` an error occurred

`NOERROR` the specified connection was marked as asynchron

Possible Errors

asynEventHandler::addAsynSender: no CEP the call to `CEPSet::getCEP` returned a `NULL` value

asynEventHandler::addAsynSender: fcntl:setown failed the system call to `fcntl` with command `setown` failed, `errno` should be checked for further informations

asynEventHandler::addAsynSender: fcntl:fasyn failed the system call to `fcntl` with command `fasyn` failed, `errno` should be checked for further informations

12.2.6 Method isAsynSender

Description Checks, if a given connection to a certain host is marked as asynchron.

Synopsis

```
#include "TWasynEventHandler.h"

bool isAsynSender( IdType possibleSender, ConnNr cn )
```

Parameters

IdType possibleSender the host to which a connection should be checked

ConnNr cn optional parameter, if omitted, connection 0 will be used

return values

false if the connection is not marked as asynchronous

true if the connection is marked as asynchron

Possible Errors

asynEventHandler::isAsynSender: no CEP the call to `CEPSet::getCEP` returned a `NULL` value

asynEventHandler::isAsynSender: fcntl:fasyn failed the system call to `fcntl` with command `fasyn` failed, `errno` should be checked for further informations

12.2.7 Method blockAsynEvents

Description Blocks the delivery of the `SIGURG` signal. If a signal occurs during a phase when it is marked as being **blocked**, it will not be delivered to the process before a call to `unblockAsynEvents` is done.
Please note, that the signal is only delayed, not lost.

Synopsis

```
#include "TWasynEventHandler.h"

static void blockAsynEvents( void )
```

Parameters

none

return values

none

Possible Errors

none

12.2.8 Method unblockAsynEvents

Description The delivery of `SIGURG` signals to the calling process is enabled again Any pending `SIGURG` signals will be delivered directly after the call is finished.

Synopsis

```
#include "TWasynEventHandler.h"

static void unblockAsynEvents( void )
```

Parameters

none

return values

none

Possible Errors

none

12.2.9 Method ignoreAsynEvents

Description If an asynchronous event from the process with the given id arrives, it will be ignored. This means that the signal is caught by the libraries signal handler, but the user handler is not invoked. Instead, the signal is discarded.

Please note, that it is not possible to ignore asynchronous events on a certain connection. A call to this method will cause asynchronous events arriving on any connection to the specified process to be ignored.

Synopsis

```
#include "TWasynEventHandler.h"

void ignoreAsynEvents( IdType from )
```

Parameters

IdType from the id of the process from which subsequent asynchronous events will be ignored

return values

none

Possible Errors

none

12.2.10 Method ignoreAsynEvents

Description If an asynchronous event from any process within the specified id group arrives, it will be ignored. This means that the signal is caught by the libraries signal handler but the user handler is not invoked. Instead, the signal is discarded. Please note, that it is not possible to ignore asynchronous events on a certain connection. A call to this method will cause asynchronous events arriving on any connection to any process with an id in the given group to be ignored.

Synopsis

```
#include "TWasynEventHandler.h"

void ignoreAsynEvents( IdGroup* from )
```

Parameters

IdGroup* from a pointer to a group of process id's from which subsequent asynchronous events will be ignored

return values

none

Possible Errors

none

12.2.11 Method allowAsynEvents

Description If asynchronous events from the process with the given id where marked as to be ignored, a call to this method will reset the situation, i. e. asynchronous events from the process with the specified id will be delivered to the user handler again.
Please note, that any asynchronous events that were sent by that process between the calls to `ignoreAsynevents` and this method are lost.

Synopsis

```
#include "TWasynEventHandler.h"

void allowAsynEvents( IdType from )
```

Parameters

IdType from the id of the process from which subsequent asynchronous events will be processed again

return values

none

Possible Errors

none

12.2.12 Method allowAsynEvents

Description If asynchronous events from the any process within the given id group where marked as to be ignored, a call to this method will reset the situation, i. e. asynchronous events from a process with an id in the group will be delivered to the user handler again.
Please note, that any asynchronous events that were sent by a process in the specified group between the calls to `ignoreAsynevents` and this method are lost.

Synopsis

```
#include "TWasynEventHandler.h"

void allowAsynEvents( IdGroup* from )
```

Parameters

IdGroup* from a pointer to group of process id's from which subsequent asynchronous events will be processed again

return values

none

Possible Errors

none

12.2.13 Method sendAsynEvent

Description Sends an asynchronous event to the process with the given id. The structure of an `event` is defined in `TWtypes.h`.

Synopsis

```
#include "TWasynEventHandler.h"

int sendAsynEvent( IdType to, event *ev )
```

Parameters

IdType to the id of the process to which an asynchronous event should be send

event* ev a pointer to an `event` structure (defined in `types.h`)

return values

`ERROR` the event could not be sent

`NOERROR` the event was successfully sent to the destination process

Possible Errors

asynEventHandler::sendAsynEvent: no CEP the call to `CEPSet::getCEP` returned a `NULL` value

asynEventHandler::sendAsynEvent: can't write OOB data the one-byte asynchronous event message could not be delivered, `errno` should be checked for further informations

asynEventHandler::sendAsynEvent: can't write event data an error occurred during the write attempt of the normal data following the asynchronous one-byte message, `errno` should be checked for further informations

12.2.14 Method sendAsynEvent

Description Sends an asynchronous event on the given connection to the process with the given id. The structure of an `event` is defined in `TWtypes.h`.

Synopsis

```
#include "TWasynEventHandler.h"

int sendAsynEvent( IdType to, ConnNr cnr, event *ev )
```

Parameters

IdType to the id of the process to which an asynchronous event should be send

ConnNr cnr the coonnection to use

event* ev a pointer to an `event` structure (defined in `types.h`)

return values

`ERROR` the event could not be sent

`NOERROR` the event was successfully sent to the destination process

Possible Errors

asynEventHandler::sendAsynEvent: no CEP the call to `CEPSet::getCEP` returned a `NULL` value

asynEventHandler::sendAsynEvent: can't write OOB data the one-byte asynchronous event message could not be delivered, `errno` should be checked for further informations

asynEventHandler::sendAsynEvent: can't write event data an error occurred during the write attempt of the normal data following the asynchronous one-byte message, `errno` should be checked for further informations

12.2.15 Method handleAsynEvents

Description This method is called (by the operating system) on every occurrence of a `SIGURG` signal, i. e. it is never called by a user program directly. It will read an asynchronous message from a connection in the associated *CEPSet* object and call the user handler if it is a user event. If it is not a user event, the method handles the event by itself and the user program is not affected in any way.

Synopsis

```
#include "TWasynEventHandler.h"

static void handleAsynEvents( void )
```

Parameters

none

return values

none

Possible Errors

asynEventHandler::AsynEventhandler: exception failed
the event handler was not able to detect on which connection an asynchronous message arrived, `errno` should be checked for further informations

asynEventHandler::handleAsynEvents: (waste)read failed there was pending normal data before the urgent message that could not be read properly, `errno` should be checked for further informations

asynEventHandler::handleAsynEvents: read failed the read operation for the `event` structure failed, `errno` should be checked for further informations

asynEventHandler::handleAsynEvents: unknown event code
the `event` structure contained a operation code which is not known by the library

asynEventHandler::handleAsynEvents: ignored asyn event from xx
this is not an error message in the normal sense, it only informs the user that an event was ignored.

12.3 Examples

12.4 Implementational Issues

In this section, we will concentrate on two aspects of the implementation of the asynchronous events in the teamwork library. First of all, we will take a look at the general aspects of urgent data, especially in a TCP/IP/UNIX environment. Then, we will explain the library specific distinction in user and system events.

First of all, what is urgent data? There is no exact, generally accepted definition of this term and therefore, our first definition is very straightforward:

Definition 12.4.1 (urgent data (general version)) *Urgent data is all kind of data which should be delivered to another process prior to any data waiting in a read queue.*

There are thousands of possibilities to implement an urgent data concept and each of them has it's pro's and con's. A presentation, even of the basic ideas, of the different approaches would blow up the framework of this report. Therefore, we will only mention a few keywords for a differentiation scheme. The interrested reader should consult, e. g, [Ste90] for a more detailed overview. Some properties of an urgent data concept are:

- to have multiple priority classes for data vs. having only "urgent" or "normal" data
- using a single connection for each class of data vs. multiplexing a single connection
- allowing more then one unread urgent message in a data stream vs. forcing an urgent message to be read before another cna be sent
- allowing messages of arbitrary length vs. a fixed size

As we have already said, these are only some ideas. There are a lot more of them, and they can all be combined in various ways. This should make clear that the number of possible implementations is countless.

Before we describe the urgent data mechanism of the TCP/IP protocol suite, we would like to extend the first definition of urgent data. This second definition is a bit more restricted towards the needs in the teamwork library.

Definition 12.4.2 (urgent data (teamwork version)) *Urgent data is all kind of data which should be delivered to another process prior to any data waiting in a read queue. Furthermore, there must exists a mechanism to inform the process which was sent urgent data about the data arrival. This informing mechanism may break the process's normal flow of operation, enabling it to handle the urgent data immediately.*

Now, let's see what the main properties of the TCP/IP urgent data implementation are. The protocol suite:

- has only two class of data – "urgent" and "normal"
- uses the same connection for both types
- allows only one outstanding message at a time
- has a fixed message length of one byte.

Especially the last point in this itemization looks very restricting, but we will see later, that this concept is absolutely sufficient to implement the libraries asynchronous event concept. Before we come to the evnt concept of the library, however, we will first present the different steps that are necessary in general to send and receive and asynchronous event.

Step 1 The receiver must mark its socket as asynchron. This is done with two calls to the `ioctl` function of the operating system.

Step 2 The receiver installs a signal handler for the `SIGURG` signal. That handler will be called when an urgent message arrives.

Step 3 The sender calls the system function `send(..., ..., ..., MSG_OOB)` to send an urgent message of one byte length to the receiver.

Step 4 When the urgent data arrives at the receiver side, the operating system calls the handler installed in Step 3. First of all, the receiver must find out on which connection the urgent data is pending. This can be done with the `select` system call. After the receiver knows on which connection the urgent data arrived, the data must be extracted from the read queue. This is due to the fact, that the operating system does not place an urgent message at the front of the read queue, it only informs the receiver that there is an urgent message somewhere in the queue. Therefore, receiver does not know whether the next `read` operation on that connection will return the urgent message, or if there is normal data before the urgent message (see figure 12.1)

To handle such a situation, the UNIX system offers a function which tells the calling process if there is normal data between the current read

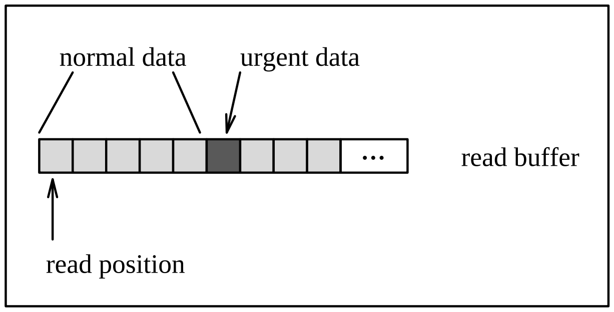

Figure 12.1: Unread normal data before pending urgent data

position and the urgent message. In this case, the next `read` call will read the entire amount of normal data, regardless which data size was specified in the call. After that read operation, the read buffer will look like figure 12.2.

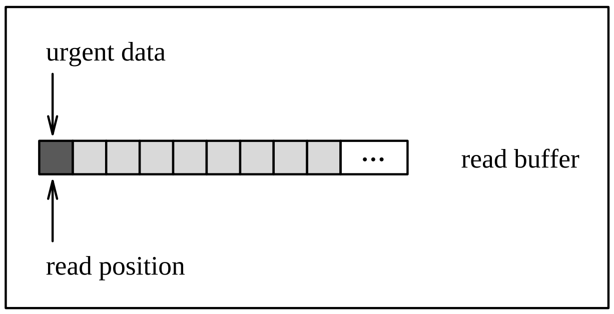

Figure 12.2: The normal data is removed

Now, the receiver can call `recv(..., ..., ..., MSG_OOB)` to read the urgent data.

Step 5 After the receiver has handled the urgent message, it leaves the signal handler and its normal execution is resumed.

This was the general approach to exchange urgent data, but there remains one big problem with it. In Step 4, when the receiver clears its read buffer up to the urgent data, the unread normal data may get lost. The solution to this problem is to save that data in a separate buffer. Any read operation would then have to look in that buffer and return the data stored there before reading data from the connection. By doing this, the data loss would be avoided, but it would produce a lot of overhead on every read operation. Since data loss because of asynchronous events is very rare, the overhead does not stand in relation to the benefit gained.

Therefore, we decided not to implement this method. Instead, we only advise the library user not to use asynchronous events during normal data exchange phases. This is not a real restriction, because the user is free to create a new *CEPSet* object and use it for asynchronous events exclusively. By doing this – at the cost of more connections between the processes involved – no data loss an occur. For the teamwork library, however, this is not needed, because we do not use asynchronous events during the team meeting (which is the only data exchange phase), or if we do, the asynchronous events used will cause a program termination and therefore, data loss does not matter.

By comparing what we said in steps 1 - 5 to the library code (methods `sendAsynEvent` and `handleAsynEvents`), it should be easy to tell how the asynchronous event exchange is done. There is only one point left which we want to explain, and this is how we cope with the limitation of a message length to one byte. The solution to this problem is quite easy: the urgent message itself does not contain any useful data, it is only sent to activate the asynchronous event handler of the receiving process. After that handler is called, it uses a `select` call to tell on which connection the urgent message arrived. That connection is then cleared by reading the "waste" (pending normal data) if necessary. Then, the sender and the receiver can communicate using normal data.

The last thing that we want to explain in this section is the distinction between two types of asynchronous events: user events and system events. Every time when an asynchronous event is read, the handler (`handleAsynEvents`) uses the `code` field of the `event` structure (defined in `types.h`) the see if it is a user or a system event. If that field contains a value of `USER_EVENT`, the user event handler is invoked (with the event code and the event parameters as arguments). If the `code` field contains another value, the system handler checks if a corresponding method is implemented. If this is the case, the method is performed and the handler returns. If no appropriate method is found, an error message is issued and the handler returns as well. The only system event which is implemented at the moment is the `PING` command. If this event is sent to a processes, a dummy message will be sent back to the sender to indicate, that the receiving process is still alive. System events can be used for an extension of the libraries functionality, the average user will not have to use them in any way.

Chapter 13

Class *timer*

13.1 General Description

This class is not included in the overview on page 6, but since it is very usefull in many situations, we have included it in our class description. The *timer* objects provided by the teamwork library fulfill two major tasks:

- they can be used as **stopwatches**, or
- they can be used as **alarm clocks**.

To use a *timer* object as a stopwatch, the user simply creates such an object, and then he can use the following methods to operate on the new object:

start To start the watch

stop To stop it

cont To restart it after it was stopped. Please note, that the times the watch was running are summed up, i. e. the watch does not start at 0 again.

elapsed This method will return the time elapsed between the first start and the last stop of the watch if it is not running. If the watch was not stopped prior to the call of this method, it will return the time elapsed between the first start and the call to the method. Please note, that the watch is not stopped in the latter case.

To use a `timer` object as alarm clock is not as easy as using it as stopwatch – it requires some user preparation. The main thing to do is to implement user timeout handler, i. e. an object which will be called when the alarm time expires. That object must be derived from class *userTimerHandler* and it should override the `timeout` member function of the base class. After an object of the new class is announced to the library as the user timeout handler, it's `timeout` method will be called when a specified amount of time expires.

In the teamwork library, we have two sorts of alarms and therefore, we also have two possibilities to announce a user handler to the system:

- A **single alarm** after the specified amount of time expires will be issued when the user handler is installed by using the `alarm` method. The user handler remains installed after the alarm was handled, but the timer is not restarted
- A **periodic alarm** can be installed by using the `periodicAlarm` method to announce a user handler to the system. With this method, the user handler is called every time the specified alarm time expires, i. e the timer is reloaded with it's initial value after the alarm is handled.

Please note, that only one user timeout handler (for either a single alarm or a periodic alarm) can be active at a time. To uninstall a user handler or to stop a periodic alarm, the member function `alarmOff` can be used.

13.2 Interface

13.2.1 Method timer

Description Constructor for an object of class *timer*. During the constructor call, the signal handler for a `SIGALRM` signal is installed.

Synopsis

```
#include "TWtimer.h"

timer( void )
```

Parameters

none

return values

none

Possible Errors

none

13.2.2 Method ~timer

Description Destructor for an object of class *timer*, the default handler for `SIGALRM` signals is set again.

Synopsis

```
#include "TWtimer.h"

~timer( void )
```

Parameters

none

return values

none

Possible Errors

none

13.2.3 Method start

Description Starts the stop watch.

Synopsis

```
#include "TWtimer.h"

void start( void )
```

Parameters

none

return values

none

Possible Errors

none

13.2.4 Method stop

Description Stops the stop watch.

Synopsis

```
#include "TWtimer.h"

void stop( void )
```

Parameters

none

return values

none

Possible Errors

none

13.2.5 Method cont

Description If the watch was stopped, it is started again. If it was not stopped, a call to this method has no effect.

Synopsis

```
#include "TWtimer.h"

void cont( void )
```

Parameters

none

return values

none

Possible Errors

none

13.2.6 Method elapsed

Description This method returns the time elapsed between the calls to `start` and `stop` if the watch is not running. Otherwise, it returns the time elapsed between the call to `start` and the call to this method.

Synopsis

```
#include "TWtimer.h"

double elapsed( void )
```

Parameters

none

return values

none

Possible Errors

none

13.2.7 Method alarm

Description Installes a user timer event handler. If the specified time expires, the handler function specified is called by the system. A new handler can **not** be installed if an older timer has not yet expired.

Synopsis

```
#include "TWtimer.h"

static int alarm( float time, timerEventHandler* Hndl )
```

Parameters

float time the time after which the user handler should be invoked in seconds

timerEventHandler* Hndl a pointer to the handler object to call when the timer expires

return values

`ERROR` if an error occurred

`NOERROR` if the user handler was installed successfully

Possible Errors

timer::alarm: error in setitimer the system call to `settimer` failed, `errno` should be checked for further informations.

timer::alarm: there is already an active timer a previously installed user handler was not yet called. The old handler can be uninstalled with a call to `alarmOff`.

13.2.8 Method periodicAlarm

Description Installes a user timer event handler. If the specified time expires, the handler function specified is called by the system. Then, the timer is reset to the initial value. By doing this, the handler function will be called periodically every n seconds. A new handler can **not** be installed if an older timer has not yet expired.

Synopsis

```
#include "TWtimer.h"

 static int periodicAlarm( float time, timerEventHandler* Hndl )
```

Parameters

float time the time after which the user handler should be invoked in seconds

timerEventHandler* Hndl a pointer to the handler object to call every time the timer expires

return values

`ERROR` if an error occurred

`NOERROR` if the user handler was installed successfully

Possible Errors

timer::periodicAlarm: error in setitimer the system call to `settimer` failed, `errno` should be checked for further informations.

timer::periodicAlarm: there is already an active timer a previously installed user handler was not yet called. The old handler can be uninstalled with a call to `alarmOff`.

13.2.9 Method alarmOff

Description Turns a previouly with with `alarm` or `periodicAlarm` installed timer off.

Synopsis

```
#include "TWtimer.h"

static int alarmOff( void )
```

Parameters

none

return values

`ERROR` if an error occurred

`NOERROR` if the user handler was uninstalled successfully

Possible Errors

timer::alarmOff: error in setitimer the system call to `settimer` failed, `errno` should be checked for further informations.

13.2.10 Method TEH

Description This method is the signal handler for the `SIGALRM` handler, it will be called by the operating system when the `SIGALRM` signal occurs. If a user handler is installed when this method is called, it will be invoked. Otherwise, an error message is issued.

Synopsis

```
#include "TWtimer.h"

static void TEH( void )
```

Parameters

none

return values

none

Possible Errors

timer::TEH: no user timer event handler installed a timer signal occurred but no user handler is installed.

13.3 Examples

13.4 Implementational Issues

The implementation of this class is not very difficult and therefore, the library code documentation should be sufficient.

Chapter 14

Class *teamworkApplication*

14.1 General Description

This class is the most complex one provided by the teamwork library, since it contains the complete framework for an entire teamwork application. The basic ideas of the teamwork approach have already been discussed in the masterthesis.

In this section, we will present the mapping of the theoretical concept to an actual implementation. The flow of control during a teamwork application is almost linear (except for one loop) and therefore, we will go through the different stages in chronological order and present the methods needed in each of the stages. It is, however, very difficult to explain this class because it is only a framework with almost no practical functionality. Therefore, we recommend to study the chapter of the masterthesis which presents the example application to see, how the ideas presented here can be applied to a real application. To attach such a real problem, the library user will normally implement his own problem specific class which is derived from class *teamworkApplication* and extend the functionality provided by the base class by the problem specific methods. These will be, in most cases, all methods that deal with the data representation (since that is problem dependent) and all methods which have to do with the physical location of resources needed (e. g. executable files).

In figure 14.1, one can see the general structure of a teamwork application. It consists of an initialization phase followed by a work/meeting-loop. The *end* sequence of an application can not be reached during the normal flow of control, it will be executed as a so called **exception** (implemented as asynchronous

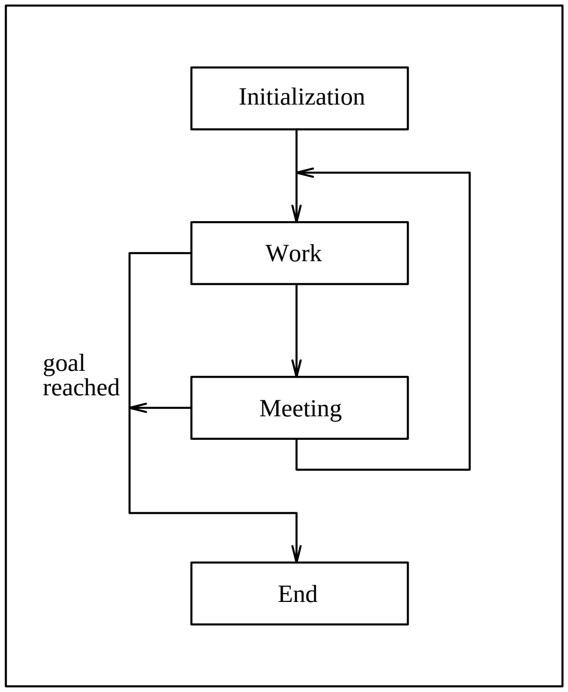

Figure 14.1: Phases of a teamwork application

event) when the goal of the application is reached by any of the processes. In the rest of this section, we will discuss the different stages shown in figure 14.1 in detail.

The execution of a teamwork application begins, when the teamwork root process is started from the UNIX command shell by the user. The user program should then instantly create a *teamworkApplication* object and pass control over to it by calling the `main` method of the new object. The `main` method takes the standard UNIX command line arguments as parameters and it implements the framework for the stages presented in figure 14.1. Therefore its first task is to perform the initialization phase. First of all, a method to parse the command line arguments specified is invoked. This is method `handleCmdLineArgs( int argc, char** argv, stringList* hosts )`. In the base classes implementation of that method, the user is queried interactively for the names of the host where the team members should be started. Therefore, the `handleCmdLineArgs(...)` method should also be overridden by the user application in order to parse the application specific command line parameters. As a result of the parse method, a list of hostnames must be returned to the calling method. That list is needed later when the remote processes are started.

In the next step, the `teamworkApplication` object will attempt to start the id server described in the masterthesis. The method `startup_startIdServer` should be overridden by the user application if the path for the id servers executable file differs from that specified in the method or if the library user whishes another output specification than the one fixed there. The implementation of the base classes member function may serve as a template for how to rewrite the method in a derived class. Before the id-server is started, an id must be assigned to the teamwork root process. This id can be chosen arbitrarily, its default value is 1. If the library user wants another id for the teamwork root, method `startup_getSVId` should be overridden in the derived class.

In the next step of the application startup sequence, the teamwork root uses the list returned by method `handleCmdLineArgs(...)` to start the remote processes. The startup sequence for each of the processes has a number of user hooks to customize the start parameters for every single host. Each of these hooks (`startup_getId, startup_getCmdLineArgs, startup_getExecfile` and `startup_getOutputSpecification`) takes a string containing the hostname as parameter and an second paramter where it may return a host specific value. In the base class implementation, these hooks are empty or they return default values. For example, the default executable file for a remote process is the same file as for the teamwork root. Please note, that the `startup_getId` hook is only executed if no id server was started.

During the startup loop (method `startProcesses`), each newly created process is assinged a unique identification number. This number is obtained by querying the id server, which guarantees, that no number is used twice during an entire teamwork application. Please note, that the id server can be used in all cases where unique identification numbers are needed, not only when a new process is created.

Since the remote processes created during the `startProcesses` method usually use the same executable file as the teamwork root, they will also create a `teamworkApplication` object and call its `main` method. That method will then determine that the process is not the teamwork root and it will therefore execute other code than the root process. Its first task is to call a member function which handles the command line arguments specified by the creating process. These parameters can be specified by the root process in method `startup_getCmdLineArgs` mentioned above, the method to handle the arguments on the child process' side is called `handleCmdLineArgs`. The base class' method does not contain any code, because the command line arguments are also application specific. After the remote processes are started by the teamwork root, the status of the application is similar to that shown in figure 14.2.

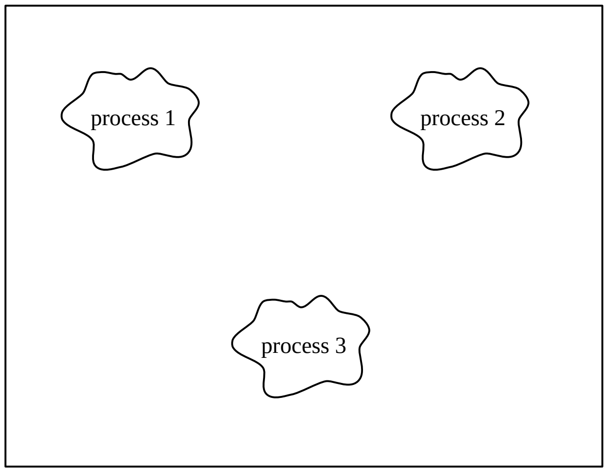

Figure 14.2: The Application Processes are started

In the figure, we have show a number of "bubbles" to indicate the processes directly after they are created in different host computers. As one can observe in figure 14.2, the processes are not connected in any way. Therefore, the next step in the startup sequence is to install the communication facilities needed in a teamwork application by calling the `setupCommunication` method. Beyond other things, that method will install a broadcast buffer and organize the asynchronous communication. After the communication network is installed, the teamwork application will look like shown in figure 14.3

After the libraries initialization phase is finished, the library user is given the chance to perform additional initializations by overriding the `initUserCode` member function. The user can, for example, create the *Xpert/Specialist/Referee* objects in that method and announce them to the *teamworkApplication* object. Also, the initial expert/referee/specialist schedule should be chosen in that method and distributed among the teamwork processes. What this schedule is and what it is used for will be explained soon.

The last thing to do during the initialization phase is to transfer the initial problem description from the teamwork root (which is also the first team supervisor) to the other team members. Therefore, the supervisor calls the `sendInitialSystem` method and the team members call their

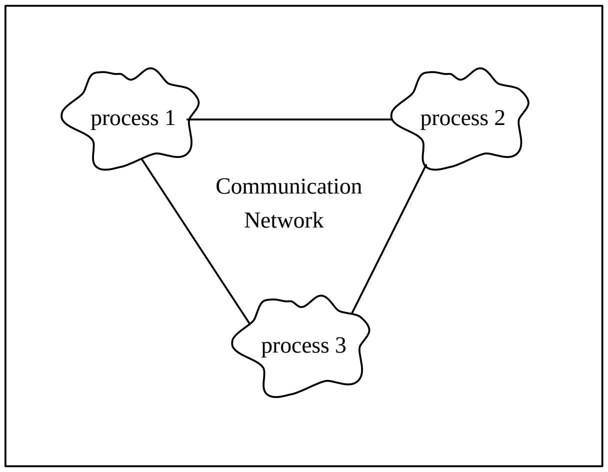

Figure 14.3: The Communication Network is installed

`receiveInitialSystem` methods to read the data sent by the supervisor. Again, these methods are problem specific and therefore, they are empty in the base class.

In table 14.1, we have summed up the different activities taken by the teamwork root and its children during the starting phase of a teamwork application.

Before we proceed with our description of the different phases of a teamwork application, we will first take a look at the main objects which interact **inside** of each process shown in figure 14.3.

In figure 14.4, one can see that each process contains four major types of objects: a number of experts, specialists, referees and a *teamworkApplication* object which controls the (process-) internal and external interaction of objects. The *teamworkApplication* object itself consists of five major parts which serve the following tasks:

- The **external communication basis** is used to communicate with other processes of the application. This kind of communication is done via sockets provided by the operating system.
- The **internal communication basis** is needed to coordinate the experts/specialists/referees and it uses internal variables to fulfill its tasks. Please note, that we use the term "internal communication" only for the communictaion between the user application and the experts/specialists/referees. The communciation of ex-

root	child
main	main
init	init
startUp_getRootId	
startUp_startIdServer	
handleCmdLineArgs	
startProcesses	startProcess
(startUp_getId)	
startUp_getCmdLineArg	
startUp_getExecfile	
startUp_getOutputType	
	handleCmdLineArgs
setupCommunication	setupCommunication
initUserCode	initUserCode
sendInitialSystem	receiveInitialSystem

Table 14.1: Summary of the start-up phase

perts/specialists/referees among themselves is no internal communication in our sense, because it is done indirectly via the application object.

- The **schedulers** decide which expert/specialist/referee to call in a process during a work/assessment phase.
- The **internal data basis** contains any informations needed by a *teamworkApplication* itself, e. g. the assignment lists described later.
- The **user data structures** are the problem dependent part of the *teamworkApplication* object. In the base class, no user data structures are defined.

The *Xpert/Specialist/Referee* objects shown in figure 14.4 implement the different algorithms used to generate new results and to asses these results, respectively. We will discuss these objects and their interaction with a *teamworkApplication* object later. First, we will explain what a expert/specialist/referee **schedule** is and how it can be controlled by the library user.

A schedule for each of the three object types (*Xpert/Specialist/Referee*) is nothing more than an associative list which assigns a expert/specialist/referee id to

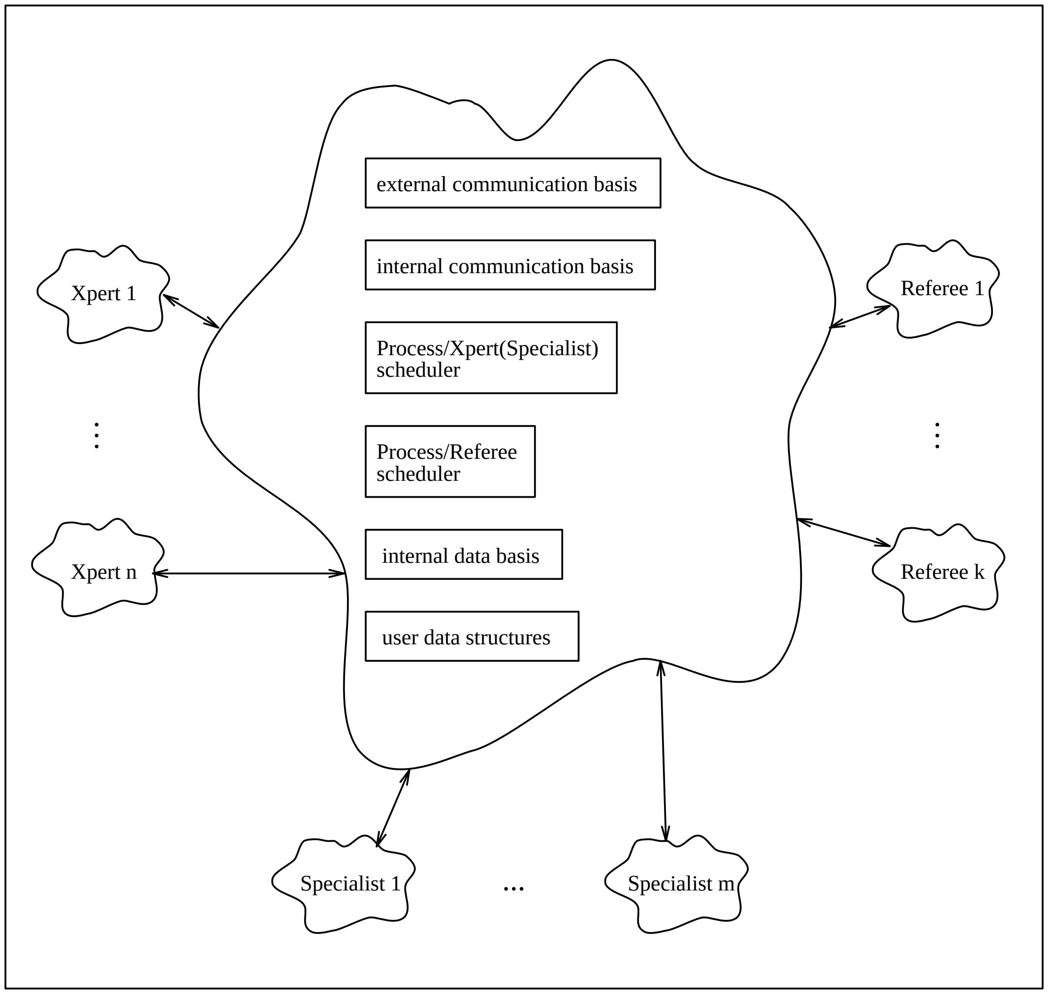

Figure 14.4: The Main Objects inside a Teamwork Process

a process id. Assuming that the library user has implemented a number of experts/specialists/referees, his first task is to announce these objects to the *teamworkApplication* object and thereby to assign a unique id to each of them. This first step is done with several calls to the `announceXXX` methods, where `XXX` stands for `Xpert, Specialist` or `Referee`. If one of the objects should be released from its id, the `removeXXX` methods can be used.

After the experts/specialists/referees are announced to the *teamworkApplication* object, the user can specify a schedule. To do this, it simply has to perform some calls to the `assignXXX` methods (where `XXX` has the same meaning as above). Each of these methods takes a process id and a *Xpert/Specialist/Referee* id as parameter. The schedules generated with these calls are stored in the `teamworkApplication` object which will use them in the `work` or `assess` methods, respectively. Usually, the schedules are created by the team supervisor and distributed to the other team members. Although the distribution of the schedules is normally done by the base class, the teamwork library provides some methods to send and receive the assignments which can be used by user programs. These methods are called `sendAssignment` and `receiveAssigment`. After this short insight in the internal structure of a process used by a teamwork application, we will return to the discussion of the different stages of such an application.

When the team members are started and the communication network is installed, the work/meeting loop begins. Before the working phase can start, the team supervisor has to determine the parameters for it by calling the member function `chooseWorkingParams`. Especially the duration of the working phase is important in this context since the team supervisor is responsible for interrupting the experts/specialists after a certain period of time. After the parameters have been fixed, the working phase begins.

To decide which expert/specialist to run, each `teamworkApplication` object has a working phase scheduler (see figure 14.4) which is called `work`. That method first checks the process/expert schedule if the user specified a expert to run in the local process. In this case, the `work` method of the specified expert is called. If no matching entry is found, the scheduler will check the process/specialist assingment. If an entry for the local process is found, the `work` method of the specialist is invoked. If no specialist is specified either, an error message is issued by the `teamworkApplication` objects work scheduler. Please note, that if both an expert and a specialist are specified for a certain process, only the experts `work` method will be executed.

Please note also, that base class does not take any influence on the duration of the working phase. It lies within the full responsibility of the library user to stop it after a certain amount of time. Usually, a `timer` object will be used to control the duration of a working phase. Refer to the code of the example application for a practical application of a `timer` object for this purpose.

As we have seen, the *Xpert/Specialist/Referee* objects are rather independent of a `teamworkApplication` object and therefore, ethods to implement some sort of communication between these object must be provided. To serve this need, each new *Xpert/Specialist/Referee* object is given a pointer to the `teamworkApplication` object which makes use of its functionality. The `teamworkApplication` object itself provides some member functions which can be used by the expert/specialist/referee to query or even manipulate the application status. These internal communication methods include, for example, the boolean function `supervisorMode` which returns `true` if the local process is the actual team supervisor and `false` otherwise. Two internal communication methods which are normally used to control the duration of the working phase are `stopWork` and `workStopped`. The second method is used by the experts/specialists to test an internal flag which indicates, that the team supervisor sent an asynchronous event to stop the working phase. The value of that flag can be set in all processes of a certain team by the supervisor with a call to `stopWork` which will issue the asynchronous event. If the `workStopped` method returns a value of `true`, the experts/specialists should instantly quit their `work` method. Another communication method which is important in this context is `goalReached`. That method should be called by an expert/specialist if he found the solution of the actual problem. The other processes of the teamwork application will then be informed that they can stop working and the application enters its end phase described later. This short summary of the internal communication methods should be sufficient for now, we will return to this topic in section **??** when we discuss the base classes for *Xpert/Specialist/Referee*. We will omit a summarization of the activities during the working phase because there are only two calls which have to be done.

After the supervisor has issued a `stopWork` event, the experts/specialists will leave their `work` method and a **team meeting** as presented in the general description begins. Since the implementation of almost all methods needed during a team meeting depends on the data structures of the user application, most methods will have to be overridden by the derived class. Only two of the base class' methods contain a little code to support the user application. All the other base class methods are empty.

The first method called at the beginning of a team meeting is one the two methods which will not have to be implemented by the derived class. That method is the scheduler for the referee assignment. The `assess` method will check the process/referee list for which referee to invoke. If no entry for the local process is found in the assignment list, an error message will be issued. If an entry is found, on the other hand, the `assess` method of the referee object assigned to the local process is called. During the assessment done by the referee, it will usually place its report in the data area of the `teamworkApplication` object. When the `assess` method of the referee object has finished its work, the `teamworkApplication` object will then invoke the `sendShortReport` (if the local process is in expert mode) or the `receiveShortreports` (if it's in supervisor mode) to transfer the reports generated by the referees to the supervisor. After the supervisor has received all (short) reports, it will decide which of the teamwork processes will be its successor. This is done in method `chooseNewSupervisor`. That method is also responsible for the distribution of the choice, the experts can receive the result of the supervisors decision during their call to `getNewSupervisor` which must return the id of the new supervisor (as must method `chooseNewSupervisor`). The old and the new team supervisor will then exchange some important informations concerning the running application (e. g. the location and the port number of the id server, the actual team members etc.), before the new supervisor calls method `receiveFullReports` to get the results which were chosen by the referees to be "good". The experts must therefore call method `sendFullReport` to transfer the (entire) assessment to the new supervisor. After the supervisor has received all the full reports, it will integrate them into its problem description. This is done in method `integrateResults`.

The next thing for the team supervisor to do, is to find a expert/specialist/referee schedule for the next work/meeting phase. The duration of the next working phase is usually also fixed during the call of this method. The decision about the structure of the team during the next working phase can have an enormous impact on the efficiency of the teamwork method. We will not go into details about planning algorithms; a good example for this complicated topic, however, can be found in [Kro95] which presents some algorithms for our distributed theorem prover DISCOUNT ([Pit93], [ADF95]).

The `changeTeamSetup` method is called by both, the supervisor and the experts and it should execute different code, depending on who has called it (see method `supervisorMode`). If it was invoked by the supervisor, the new schedule must be made and it must be sent to the experts. The experts, on the other hand, must call some methods to receive the schedule fixed by the supervisor. The base class' implementation of this method offers some user support to send the schedule from the supervisor to the experts. After the supervisor has done its `assignXXX` (see above) calls, it can invoke the base class' `changeTeamSetup` method which will then send the schedule to all processes currently in the team. Of course, the experts must also call the base class' method in order to receive the schedule sent by the supervisor.

After the schedule for the next work/meeting phase is distributed among the team members, the supervisor calls `sendNewSystem` to transfer his problem description to the experts, which themselves call method `receiveNewSystem`. When the transfer phase for the new problem description is finished, a new cycle begins. In table 14.1, we have summarized the different actions for the supervisor and the normal team members during a team meeting.

Supervisor	Team member
assess	assess
receiveShortReports	sendShortReport
chooseNewSupervisor	getNewSupervisor
new Supervisor	**Team member**
receiveFullReports	sendFullReport
integrateResults	
changeTeamSetup	changeTeamSetup
sendNewSystem	receiveNewSystem

Table 14.2: Summary of the team meeting activities

The last stage of a teamwork application is entered when at least one of the experts has reached the goal. If two or more experts find the solution almost simultaneously, only one will be able to announce this to the other team members. It depends on some operating system internal effects which of the processes it will be. This is not a real shortcoming of the implementation, since it is usually a very rare case that two processes reach the goal at the same time. Also, the user will normally be interrested in only one solution.

The second possiblity when the goal of a search process can be reached is during a team meeting (see figure 14.1). The only process which is able to detect that a solution was found during a meeting is the actual supervisor. This situation can occur, e. g., when the supervisor integrates the good results of the experts into it's own problem description and then recognizes, that the experts results together with it's own data form a solution to the problem.

To inform the processes of a teamwork application that a solution was found, the expert who found it will call the member function `goalReached` from inside its `work` method. The `goalReached` method will issue an asynchronous event with opcode `GOAL_REACHED`. That event will cause the processes to call the `cleanup` method of the `teamworkApplication` object. That method is also called by the process which found the solution. During the execution of the `cleanup` method, the variable `reachedGoal`, located in the data area of the *teamworkApplication* object, can be used to test if the method is called from within the process which actually found the solution. Of course, only the process which has the flag set should present the solution to the user. Please note, that, since the `cleanup` method is called by the asynchronous event handler, it is **not** possible to interrupt it by subsequent asynchronous events. A normal point-to-point, or even a broadcast communication, is still possible, although it will only be needed in very rare cases. Please note also, that the `cleanup` member function will `not` return to normal program execution. Instead, the operating system function `exit` is called by the asynchronous event after the `cleanup` method is finished.

What we have skipped until now – although we took it for granted – are the communication methods used by the `teamworkApplication` objects on the respective host computers to exchange data among each other. Each of these objects contains a *processGroup* object (see section **??**) to deal with the external communication issues. In figure **??** on page **??**, the *processGroup* is referred to as the external communication basis. This object can be used for all communication issues, the user interface for that class can be found in section **??**

This overview over class *teamworkApplication* should be sufficient for the moment. The reader interrested in details should directly consult the libraries source code. As we have already said at the beginning of this section, it is very hard to present the functionality of this class without a concrete application. Therefore, we recommend to study the development of the example application in the general description which will apply the abstract concepts presented here to a real implementation.

14.2 Interface

14.2.1 Method processEvent

Description This method is the asynchronous event handler of class *teamworkApplication*. It is called by the libraries asynchronous event handler.

Synopsis

```
#include "teamwork.h"

void processEvent( usrEventCode usr_code, char *param )
```

Parameters

usrEventCode usr_code The event code of the asynchronous event which caused the handler to be invoked

char* param the parameters for that event

return values

none

Possible Errors

none

14.2.2 Method abnormalTermination

Description The fatal hook of class *teamworkApplication*. This method is executed when one of the `fatalXXX` methods of the libraries error handler `ERR` is called. This will be the last portion of user code which will be executed before the process terminates.

Synopsis

```
#include "teamwork.h"

void abnormalTermination( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.3 Method getExecfile

Description Returns the name of the currently executed file.

Synopsis

```
#include "teamwork.h"

void getExecfile( char* ef )
```

Parameters

char* ef result parameter for the filename

return values

none

Possible Errors

none

14.2.4 Method getSV

Description Returns the id of the actual supervisor.

Synopsis

```
#include "teamwork.h"

virtual IdType getSV( void )
```

Parameters

none

return values

the supervisors id

Possible Errors

none

14.2.5 Method announceXpert

Description Announce an *Xpert* object to the teamwork application and assign an id to it. If an expert with the given id already exists, it is replaced by the new object.

Synopsis

```
#include "teamwork.h"

virtual void announceXpert( IdType id, Xpert* xp )
```

Parameters

IdType id the id which should be assigned to the *Xpert* object

Xpert* xp a pointer to an *Xpert* object

return values

none

Possible Errors

none

14.2.6 Method removeXpert

Description Remove the specified expert from the list of experts known by the teamwork application.

Synopsis

```
#include "teamwork.h"

virtual void removeXpert( IdType id )
```

Parameters

IdType id the id of the expert to remove

return values

none

Possible Errors

none

14.2.7 Method announceSpecialist

Description Announce a *Specialist* object to the teamwork application and assign an id to it. If a specialist with the given id already exists, it is replaced by the new object.

Synopsis

```
#include "teamwork.h"

virtual void announceSpecialist( IdType id, Specialist* sp )
```

Parameters

IdType id the id which should be assigned to the *Specialist* object

Specialist* sp a pointer to an *Specialist* object

return values

none

Possible Errors

none

14.2.8 Method removeSpecialist

Description Remove the specified specialist from the list of specialists known by the teamwork application.

Synopsis

```
#include "teamwork.h"

virtual void removeSpecialist( IdType id )
```

Parameters

IdType id the id of the specialist to remove

return values

none

Possible Errors

none

14.2.9 Method announceReferee

Description Announce a *Referee* object to the teamwork application and assign an id to it. If a referee with the given id already exists, it is replaced by the new object.

Synopsis

```
#include "teamwork.h"

virtual void announceReferee( IdType id, Referee* ref )
```

Parameters

IdType id the id which should be assigned to the *Referee* object

Referee* sp a pointer to an *Refree* object

return values

none

Possible Errors

none

14.2.10 Method removeRefree

Description Remove the specified referee from the list of referees known by the teamwork application.

Synopsis

```
#include "teamwork.h"

virtual void removeReferee( IdType id )
```

Parameters

IdType id the id of the referee to remove

return values

none

Possible Errors

none

14.2.11 Method assignXpert

Description Make an entry in the process/expert list. The experts `work` method will be called during the execution of the `work` method of the *teamworkApplication* object if the local process' id matches an id in the process/expert list.

If an entry in the process/expert list already exists for the specified process, it is replaced by the new assignment.

Synopsis

```
#include "teamwork.h"

virtual void assignXpert( IdType process, IdType xpert )
```

Parameters

IdType process the id of the process which should run a certain expert

IdType xpert the id of the expert to run during the work phase

return values

none

Possible Errors

none

14.2.12 Method assignSpecialist

Description Make an entry in the process/specialist list. The specialists `work` method will be called during the execution of the `work` method of the *teamworkApplication* object if the local process' id matches an id in the process/specialist list.

If an entry in the process/specialist list already exists for the specified process, it is replaced by the new assignment.

Synopsis

```
#include "teamwork.h"

virtual void assignSpecialist( IdType process, IdType spec )
```

Parameters

IdType process the id of the process which should run a certain specialist

IdType xpert the id of the specialist to run during the work phase

return values

none

Possible Errors

none

14.2.13 Method assignReferee

Description Make an entry in the process/referee list. The referees `assess` method will be called during the execution of the `assess` method of the *teamworkApplication* object if the local process' id matches an id in the process/referee list.

If an entry in the process/referee list already exists for the specified process, it is replaced by the new assignment.

Synopsis

```
#include "teamwork.h"

virtual void assignReferee( IdType process, IdType ref )
```

Parameters

IdType process the id of the process which should run a certain referee

IdType xpert the id of the referee to run during the assessment phase

return values

none

Possible Errors

none

14.2.14 Method teamworkApplication

Description Constructor for an object of class *teamworkApplication*. The data structures are created and some internal variables are initialized.

Synopsis

```
#include "teamwork.h"

teamworkApplication( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.15 Method ~teamworkApplication

Description Destructor for an object of class *teamworkApplication*.

Synopsis

```
#include "teamwork.h"

virtual ~teamworkApplication( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.16 Method sendAssignment

Description Send the specified assignment list to the process with the given id.

Synopsis

```
#include "teamwork.h"

virtual int sendAssignment( IdType to, Pairset<IdType, IdType>* set )
```

Parameters

IdType to the id of the process to which the list should be send

Pairset¡IdType, IdType¿* set a pointer to the list to send

return values

`ERROR` if the list could not be packed propperly

the status of the send operation, see `processGroup::write( Idtype,... )` else

Possible Errors

teamworkApplication::sendAssignment: error in packing set the `pack` method of the *Pairset* object failed

14.2.17 Method sendAssignment

Description Send the specified assignment list to all processes in the given group.

Synopsis

```
#include "teamwork.h"

virtual int sendAssignment( IdGroup* gr, Pairset<IdType, IdType>* set )
```

Parameters

IdGroup* gr a pointer to the list of id's of the processes to which the list should be send

Pairset¡IdType, IdType¿* set a pointer to the list to send

return values

`ERROR` if the list could not be packed propperly

the status of the send operation, see `processGroup::write( IdGroup*,... )`
else

Possible Errors

teamworkApplication::sendAssignment: error in packing set
the `pack` method of the *Pairset* object failed

14.2.18 Method recvAssignment

Description Reads an assignment list from the process with the given id and stores it in the specified location.

Synopsis

```
#include "teamwork.h"

virtual int recvAssignment( IdType from, Pairset<IdType, IdType>* set )
```

Parameters

IdType from the id of the process which sends the assignment list

Pairset¡IdType, IdType¿* set a pointer to *Pairset* object where the list should be stored

return values

`ERROR` if the list could not be unpacked propperly

the status of the receive operation, see `processGroup::read( IdType,... )`
else

Possible Errors

teamworkApplication::recvAssignment: error in unpacking set
the `unpack` method of the *Pairset* object failed

14.2.19 Method main

Description The main loop of a teamwork application. See the section 14 for details.

Synopsis

```
#include "teamwork.h"

virtual void main( int argc, char **argv )
```

Parameters

int argc standard UNIX/C parameter counter

char argv** standard UNIX/C parameter array

stringList* conf_files a list of names of configuration files

return values

none

Possible Errors

none

14.2.20 Method initXpert

Description Startup sequence for processes which are not the root process.

Synopsis

```
#include "teamwork.h"

virtual void initXpert( int argc, char **argv )
```

Parameters

int argc standard UNIX/C parameter counter

char argv** standard UNIX/C parameter array

return values

none

Possible Errors

none

14.2.21 Method initSupervisor

Description Startup sequence for the root process.

Synopsis

```
#include "teamwork.h"

virtual void initSupervisor( int argc, char **argv );
```

Parameters

int argc standard UNIX/C parameter counter

char argv** standard UNIX/C parameter array

return values

none

Possible Errors

none

14.2.22 Method startUp_getRootId

Description Return the id for the root process needed in the startup sequence. This method should be overloaded if the user wnats the root process to have another id as the default value of 1.

Synopsis

```
#include "teamwork.h"

virtual IdType startUp_getRootId( void )
```

Parameters

none

return values

1

Possible Errors

none

14.2.23 Method startUp_startIdServer

Description Starts the id server with the specified process as owner. The executable file of the id server is searched in the `bin` directory specified at the compile time of the library. The name of the executable file is also fixed when the library is installed. This method should be overloaded by the user if the executable file is not located in that directory or if it has another name then the default value fixed upon compilation.

Synopsis

```
#include "teamwork.h"

virtual void startUp_startIdServer( IdType owner )
```

Parameters

IdType owner the id of the calling process

return values

none

Possible Errors

none

14.2.24 Method handleCmdLineArgs

Description This method should be overloaded by the user to parse the command line arguments for the teamwork root process. The second parameter is a result parameter which must return a list of hostnames on which the teamwork application processes should be started. If the base class method is called without any arguments, the user is queried interactively for a list of hostnames. The resulting list is needed by method 'startProcesses'

Synopsis

```
#include "teamwork.h"

virtual void handleCmdLineArgs( int argc, char** argv, stringList* hosts )
```

Parameters

int argc standard UNIX/C parameter counter

char argv** standard UNIX/C parameter array

stringList* hosts result parameter. In this varibale, the method must return a list of hostnames on which the teamwork application processes should run. The list is needed later when the remote processes are started.

return values

none

Possible Errors

Don't know what to do with the arguments, sorry... this is only a user information message. It is issued if the argument list passed to the base class' method is not empty.

14.2.25 Method startProcesses

Description Starts teamwork application processes on a number of host computers.

Synopsis

```
#include "teamwork.h"

virtual void startProcesses( stringList* hosts )
```

Parameters

stringList* hosts a list of hostname on which the teamwork application processes should be started. This is usually the list returned by the `handleCmdLineArgs( int argc, char** argv, stringList* hosts )` member function.

return values

none

Possible Errors

none

14.2.26 Method startUp_getId

Description This method checks if a resource with name ¡host*Id¿ is stores in the resource database. The name of the resource is <host>*Id.

Synopsis

```
#include "teamwork.h"

virtual IdType startUp_getId( string host )
```

Parameters

string host the name of the host on which the process for which an id should be returned is running

return values

the value of the host specific resource if it is set

INVALID otherwise

Possible Errors

none

14.2.27 Method startUp_getCmdLineArg

Description This method returns the host specific command line arguments for the teamwork process on that host if they are stored in the resource database. If none are stored there, an empty string is returned. The name of the resource is <host>*CmdLineArgs.

Synopsis

```
#include "teamwork.h"

virtual void startUp_getCmdLineArg( string host, char* arg )
```

Parameters

string host the name of the host on which the process for which the command line arguments should be returned is running

char* arg result parameter for the command line arguments

return values

none

Possible Errors

none

14.2.28 Method startUp_getExecfile

Description This method returns a host specific path for the executable file for the teamwork process on that host if one is stored in the resource database. If none is stored there, an empty string is returned. The name of the resource is `<host>*Execfile`.

Synopsis

```
#include "teamwork.h"

virtual void startUp_getExecfile( string host, char* exec )
```

Parameters

string host the name of the host on which the process for which the executable file path should be returned is running

char* exec result parameter for the executable file path

return values

none

Possible Errors

none

14.2.29 Method startUp_getOutputType

Description This method returns the `<host>*OutputType` and `<host>*OutputParam` resources for the specified host. Default values are `XTERM_OUTPUT` and an empty parameter string.

Synopsis

```
#include "teamwork.h"

virtual void startUp_getOutputType( string host, outputType* mode, char* modeParam )
```

Parameters

string host the name of the host on which the process for which the output specificationshould be returned is running

outputType* mode result parameter for the output type

char* modeParam result parameter for the output mode parameters

return values

none

Possible Errors

none

14.2.30 Method startProcess

Description Basic startup tasks for a process which is not the teamwork application root.

Synopsis

```
#include "teamwork.h"

virtual void startProcess( int argc, char **argv )
```

Parameters

int argc standard UNIX/C parameter counter

char argv** standard UNIX/C parameter array

return values

none

Possible Errors

none

14.2.31 Method handleCmdLineArgs

Description This method should be overloaded by the user to handle command line parameters passed by the root process to a child.

Synopsis

```
#include "teamwork.h"

virtual void handleCmdLineArgs( int argc, char** argv )
```

Parameters

int argc standard UNIX/C parameter counter

char argv** standard UNIX/C parameter array

return values

none

Possible Errors

none

14.2.32 Method setupCommunication

Description Installs a broadcast buffer for the teamwork root and its children.

Synopsis

```
#include "teamwork.h"

virtual void setupCommunication( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.33 Method initUserCode

Description This method should be overloaded by the user to do problem specific initializations **after** the initialization phase of the base class has completed.

Synopsis

```
#include "teamwork.h"

virtual void initUserCode( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.34 Method sendInitialSystem

Description This method should be overloaded by the user to send the initial problem description from the first supervisor to the team members.

Synopsis

```
#include "teamwork.h"

virtual void sendInitialSystem( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.35 Method receiveInitialSystem

Description This method should be overloaded by the user to receive the initial problem description from the first supervisor.

Synopsis

```
#include "teamwork.h"

virtual void receiveInitialSystem( void )
```

Parameters

none

return values

none

Possible Errors

None

14.2.36 Method setWorkingParams

Description This method should be overloaded by the user to set the problem specfic parameters for the working phase, e. g. the duration. It is called by the supervisor before the actual working phase starts.

Synopsis

```
#include "teamwork.h"

virtual void setWorkingParams( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.37 Method work

Description This method implements the scheduler for the process/expert and process/sepcialist lists. If no entry for the local process is found in the process/expert list, the scheduler will check the process/specialist list for a matching entry. If none is found there either, an error message is issued.

Synopsis

```
#include "teamwork.h"

virtual void work( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.38 Method workStopped

Description This method returns the value of the `stop_work` flag used to communicate with the asynchronous event handler.

Synopsis

```
#include "teamwork.h"

virtual bool   workStopped( void )
```

Parameters

none

return values

the value of the flag

Possible Errors

none

14.2.39 Method supervisorMode

Description Checks, if the local process is the actual team supervisor.

Synopsis

```
#include "teamwork.h"

virtual bool   supervisorMode( void )
```

Parameters

none

return values

false the local process is not team supervisor

true else

Possible Errors

teamworkApplication::supervisorMode: inconsistent specifications, assuming expert mode
the `role` attribute and the `supervisor` attribute do not report the same status. This can be due to some inconsistency in varibale settings while changing modes. In this case, expert mode is assumed.

14.2.40 Method stopWork

Description This method is called by the team supervisor to indicate the end of the actual working phase. Plase note, that only the supervisor is allowed to call this method.

Synopsis

```
#include "teamwork.h"

virtual void stopWork( void )
```

Parameters

none

return values

none

Possible Errors

teamworkApplication::stopWork: not supervisor!!!
the method was called by a process which is not the actual team supervisor. In this case, no message is send to the other processes.

14.2.41 Method goalReached

Description This method should be called by any of the epxerts/specialists to inform the other processes of a teamwork application that a solution to the problem was found. The program will be terminated after this method has completed.

Synopsis

```
#include "teamwork.h"

virtual void goalReached( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.42 Method meeting

Description This method implements the framework for a team meeting. See the section 14 for details.

Synopsis

```
#include "teamwork.h"

virtual void meeting( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.43 Method assess

Description This method implements the scheduler for the process/referee list. If no entry for the local process is found in the list, an error message is issued.

Synopsis

```
#include "teamwork.h"

virtual void assess( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.44 Method sendShortReport

Description This method should be overloaded by the user to send a short report from a normal team member to the supervisor.

Synopsis

```
#include "teamwork.h"

virtual void sendShortReport( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.45 Method receiveShortReports

Description This method should be overloaded by the user to receive a short report from a normal team member.

Synopsis

```
#include "teamwork.h"

virtual void receiveShortReports( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.46 Method chooseNewSupervisor

Description This method should be overloaded by the user to determine the next supervisor during a team meeting. This method is called by the actual supervisor. The id of the new supervisor must be brought to the team members in a way, that they can return it during their call to `getNewSupervisor`.

Synopsis

```
#include "teamwork.h"

virtual IdType chooseNewSupervisor( void )
```

Parameters

none

return values

`INVALID`

Possible Errors

none

14.2.47 Method getNewSupervisor

Description This method should be overloaded by the user to return the id of the next supervisor. This method is called by the all team members except the current supervisor. Since the new supervisor is determined by the old supervisor during his call to `chooseNewSupervisor`, some sort of data exchange will have to take place between the supervisor and the team members.

Synopsis

```
#include "teamwork.h"

virtual IdType getNewSupervisor( void )
```

Parameters

none

return values

—tt INVALID

Possible Errors

none

14.2.48 Method sendFullReport

Description This method should be overloaded by the user to send a full report from a normal team member to the supervisor.

Synopsis

```
#include "teamwork.h"

virtual void sendFullReport( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.49 Method receiveFullReports

Description This method should be overloaded by the user to receive a full report from a normal team member.

Synopsis

```
#include "teamwork.h"

virtual void receiveFullReports( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.50 Method integrateResults

Description This method should be overloaded by the user to integrate the results chosen by the referees into the problem description of the supervisor.

This method is only called by the actual supervisor.

Synopsis

```
#include "teamwork.h"

virtual void integrateResults( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.51 Method changeTeamSetup

Description This method should be overloaded by the user to determine the process/expert, process/specialist and process/referee assignment. The base class' method can be called to send and receive the new schedule.

Synopsis

```
#include "teamwork.h"

virtual void changeTeamSetup( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.52 Method sendNewSystem

Description This method should be overloaded by the user to send the problem description of the supervisor to the normal team members.

Synopsis

```
#include "teamwork.h"

virtual void sendNewSystem( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.53 Method receiveNewSystem

Description This method should be overloaded by the user to receive the problem description of the supervisor.

Synopsis

```
#include "teamwork.h"

virtual void receiveNewSystem( void )
```

Parameters

none

return values

none

Possible Errors

none

14.2.54 Method cleanup

Description This method is called asynchronously if one of the teamwork application processes has found a solution. This is the last code sequence that will be executed before the program exists. The `goalReached` predicate can be used to test if the expert/specialist running on the local process found the solution. This method should be overloaded by the user to present the solution found by an expert/specialist and to clean the environment before the teamwork application is terminated.

Synopsis

```
#include "teamwork.h"

virtual void cleanup( void )
```

Parameters

none

return values

none

Possible Errors

none

14.3 Examples

14.4 Implementational Issues

We have already mentioned, that a number of asynchronous events are needed during a teamwork application. This sort of communication is supported by he *processGroup*, the events implemented in the base class are as follows:

- `STOP_WORK`. This event code is used by the team supervisor to indicate the end of the current working phase.
- A `GOAL_REACHED` event is issued by an expert if he found a solution to the actual problem.

- `TERMINATE`. This code will cause the receiving process to terminate prematurely.

The library user can define additional asynchronous events and handle them in his own version of the `processEvent` method. It is, however, important that a user handler calls the *teamworkApplication* handler if an event with an unknow code (unknow to the user handler!) occurs. This will usually be done in the `default` action of the `switch` statement used to dispatch the user events. If the call to the base class' handler is omitted or forgotten, the base class will not be able to work correctly any longer.

The last thing we want to mention in this section are some internal variables of class *teamworkApplication*. These variables can be useful when the user would like to extend the base class' functionality. The variables are all declared as `protected` so that they can be used by the derived classes. The most interresting internal attributes are:

- The `pg` variable contains a pointer to the *processGroup* object used to represent the current team.
- `conn` is a pointer to the set of connections the local process has to others. This object can be used if data exchange with processes which are not in the process group becomes necessary.
- The `brbuf` variable contains a pointer to the broadcast buffer of the current team.
- `execfile` holds the path and filename of the executable file for the local process.
- `id_d` is a pointer to the object representing the id server.
- Finally, the `thisProcess` variable may be the most frequently used attribute, since it contains a pointer to the object representing the local process.

Bibliography

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A system for distributed equational deduction. *Proc. 6th RTA*, pages 397–402, 1995.

[BT90] Falko Bause and Wolfgang Tölle. *C++, Version 3*. Vieweg Verlag, Braunschweig, 1990.

[Com88] Douglas E. Comer. *Principles, Protocols and Architecture*. Internetworking with TCP/IP. Prentice Hall, Englewood Cliffs, NJ, 1988.

[CS93] Douglas E. Comer and David C. Stevens. *Client/Server Programming and Applications (BSD Socket Version)*. Internetworking with TCP/IP. Prentice Hall, Englewood Cliffs, NJ, 1993.

[Den93] Jörg Denzinger. *Teamwork: Eine Methode zum Entwurf verteilter, wissensbasierter Theorembeweiser (in German)*. PhD thesis, Universität Kaiserslautern, 1993.

[Den94] Jörg Denzinger. The teamwork approach to distributed search. Technical report, Universität Kaiserslautern, 1994.

[Jal90] Pankaj Jalote. *An Integrated Approach to Software Engineering*. Springer, New York, 1990.

[Kie94] Markus Kiefer. Zufallszahlengeneratoren für Parallelrechner (in German). Seminar: Moderne numerische Algorithmen, Universität Kaiserslautern, 1994.

[Kög95] Christoph Kögl. Verteilte Berechnung von Gröbnerbasen unter Verwendung des Teamwork-Paradigmas (in german). Master's thesis, Universität Kaiserslautern, 1995.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Kro95] Martin Kronenburg. Hierarchisierung der Teamwork-Methode mittels Zerlegungsplanung (in german). Master's thesis, Universität Kaiserslautern, 1995.

[Leo95] Thomas Leopold. Verteilte Lösung des Travelling-Salesman-Problems durch TEAMWORK (in german). Master's thesis, Universität Kaiserslautern, 1995.

[Lin92] Jürgen Lind. Sicheres Broadcasting für DISCOUNT (in german). Projektarbeit, Universität Kaiserslautern, 1992.

[Pit93] Werner Pitz. Realisierung eines Systems zum verteilten, wissensbasierten Gleichheitsbeweisen mit Hilfe der Teamwork-Methode (in german). Master's thesis, Universität Kaiserslautern, 1993.

[Rob65] J.A. Robinson. A Machine Oriented Logic based on the Resolution Principle. *Journal of ACM 12*, 1:23–41, 1965.

[SK90] Martin Schader and Stefan Kuhlins. *Programmieren in C++*. Springer Verlag, Berlin, 1990.

[Smu68] R. Smullyan. *First-Order Logic*. Springer, 1968.

[Ste90] W. Richard Stevens. *UNIX network Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, Massachusetts, 1992.

[Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Massachusetts, 1987.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, 1994.

[SUN90a] SUN Microsystems, Inc. *Network Programming Guide*, 1990.

[SUN90b] SUN Microsystems, Inc. *Sun OS Reference Manual Volume I*, 1990.

[SUN90c] SUN Microsystems, Inc. *Sun OS Reference Manual Volume II*, 1990.

[SUN90d] SUN Microsystems, Inc. *Sun OS Reference Manual Volume III*, 1990.

[SUN90e] SUN Microsystems, Inc. *System & Network Administration*, 1990.

[Tan81] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992.